TASKING®

TASKING VX-toolset for TriCore User Guide

Table of Contents

Chapter 1. C Language

This chapter describes the target specific features of the C language, including language extensions that are not standard in ISO-C. For example, pragmas are a way to control the compiler from within the C source.

The TASKING VX-toolset for TriCore® C compiler fully supports the ISO-C standard and adds extra possibilities to program the special functions of the target.

In addition to the standard C language, the compiler supports the following:

- extra data types, like `__fract`, `__laccum` and `__packb`

- keywords to specify memory types for data and functions

- attribute to specify alignment and absolute addresses

- intrinsic (built-in) functions that result in target specific assembly instructions

- pragmas to control the compiler from within the C source

- predefined macros

- the possibility to use assembly instructions in the C source

- keywords for inlining functions and programming interrupt routines

- libraries

All non-standard keywords have two leading underscores (__).

In this chapter the target specific characteristics of the C language are described, including the above mentioned extensions.

1.1. Data Types

The C compiler supports the ISO C99 defined data types, and additionally fractional types and packed data types. The sizes of these types are shown in the following table.

C Type	Size	Align	Limits
_Bool	1	8	0 or 1
signed char	8	8	$[-2^7, 2^7-1]$
unsigned char	8	8	$[0, 2^8-1]$
short	16	16	$[-2^{15}, 2^{15}-1]$
unsigned short	16	16	$[0, 2^{16}-1]$
int	32	16	$[-2^{31}, 2^{31}-1]$
unsigned int	32	16	$[0, 2^{32}-1]$

C Type	Size	Align	Limits
enum [1]	8 16 32	8 16	$[-2^7, 2^7-1]$ or $[0, 2^8-1]$ $[-2^{15}, 2^{15}-1]$ or $[0, 2^{16}-1]$ $[-2^{31}, 2^{31}-1]$
long	32	16	$[-2^{31}, 2^{31}-1]$
unsigned long	32	16	$[0, 2^{32}-1]$
long long	64	32	$[-2^{63}, 2^{63}-1]$
unsigned long long	64	32	$[0, 2^{64}-1]$
_Float16 (10-bit mantissa) [2]	16	16	[-65504.0F,-6.103515625E-05] [+6.103515625E-05,+65504.0F]
float (23-bit mantissa)	32	16	[–3.402E+38, –1.175E-38] [+1.175E-38, +3.402E+38]
double long double (52-bit mantissa)	64	32	[-1.797E+308, -2.225E-308] [+2.225E-308, +1.797E+308]
_Imaginary float	32	16	[–3.402E+38i, –1.175E-38i] [+1.175E-38i, +3.402E+38i]
_Imaginary double _Imaginary long double	64	32	[-1.797E+308i, -2.225E-308i] [+2.225E-308i, +1.797E+308i]
_Complex float	64	32	real part + imaginary part
_Complex double _Complex long double	128	32	real part + imaginary part
pointer to data or function	32	32	$[0, 2^{32}-1]$
struct/union [3]	>= 64	32	$[0, 2^{32}-1]$
__sfract	16	16	[-1, 1>
__fract	32	32	[-1, 1>
__laccum	64	32	[-131072,131072>
__packb signed __packb	32	16	4x: $[-2^7, 2^7-1]$
unsigned __packb	32	16	4x: $[0, 2^8-1]$
__packhw signed __packhw	32	16	2x: $[-2^{15}, 2^{15}-1]$
unsigned __packhw	32	16	2x: $[0, 2^{16}-1]$

[1] When you use the `enum` type, the compiler will use the smallest suitable integer type (`char`, `unsigned char`, `short`, `unsigned short` or `int`), unless you use C compiler option **--integer-enumeration** (always use 32-bit integers for enumeration).

[2] The C compiler supports half-precision (16-bit) floating-point via the `_Float16` type using the binary16 interchange format. See also Section 1.1.1, *Half Precision Floating-Point*.

[3] Structures and unions that are equal to or larger than 64-bit, are word aligned to allow efficient copy through LD.D and ST.D instructions. See also Section 1.1.3, *Packed Data Types*.

__bitsizeof() operator

The `sizeof` operator always returns the size in bytes. Use the `__bitsizeof` operator in a similar way to return the size of an object or type in bits.

```
__bitsizeof( object | type )
```

1.1.1. Half Precision Floating-Point

The TASKING C compiler supports half precision (16-bit) floating-point via the `_Float16` type using the binary16 interchange format. The binary16 interchange format is defined in IEEE Std 754-2008 IEEE Standard for Floating-Point Arithmetic. The `_Float16` type is defined in *ISO/IEC TS 18661-3 Draft Technical Specification – December 4, 2014 WG14 N1896*.

The `_Float16` type with binary16 format can represent normalized values in the range of 2^{-14} to 65504. There are 11 bits of significant precision, approximately 3 decimal digits. Also subnormal values are supported, as defined by `FLT16_HAS_SUBNORM` in `float.h`.

The `_Float16` type is a storage format only. For purposes of arithmetic and other operations, `_Float16` values in C expressions are automatically promoted to `float`.

Note that all conversions from and to `_Float16` involve an intermediate conversion to `float`. Because of rounding, this can sometimes produce a different result than a direct conversion.

When you specify C compiler option **--fp-model=-soft**, the C compiler generates hardware floating-point instructions for conversions between `_Float16` and `float` for AURIX PLUS only.

Language-level support for the `_Float16` data type is independent of whether the C compiler generates code using hardware floating-point instructions or not. In cases where hardware support is not specified or not available for the selected core, the C compiler implements conversions between `_Float16` and `float` values as run-time library calls. These run-time functions are called `__f_ftohp` and `__f_hptof`.

```
_Float16 __f_ftohp( float f ); // single precision to half precision
float __f_hptof( _Float16 f ); // half precision to single precision
```

1.1.2. Fractional Types

The TASKING C compiler fully supports fractional data types which allow you to use normal expressions:

```
__fract f, f1, f2;  /* Declaration of fractional variables    */

f1 = 0.5;                /* Assignment of a fractional constants  */
f2 = 0.242;

f = f1 * f2;             /* Multiplication of two fractionals     */
```

The `__sfract` type has 1 sign bit + 15 mantissa bits. The `__fract` type has 1 sign bit + 31 mantissa bits. The `__laccum` type has 1 sign bit + 17 integral bits + 46 mantissa bits.

> The `__accum` type is only included for compatibility reasons and is mapped to `__laccum`.

The TriCore instruction set supports most basic operations on fractional types directly. To obtain more portable code, you can use several intrinsic functions that use fractional types. Fractional values are automatically saturated.

Section 1.11.5, *Intrinsic Functions* explains intrinsic functions. Section 1.11.5.2, *Fractional Arithmetic Support* lists the intrinsic functions.

Promotion rules

For the three fractional types, the promotion rules are similar to the promotion rules for `char`, `short`, `int`, `long` and `long long`. This means that for an operation on two different fractional types, the smaller type is promoted to the larger type before the operation is performed.

When you mix a fractional type with a `float` or `double` type, the fractional number is first promoted to `float` respectively `double`.

When you mix an integer type with the `__laccum` type, the integer is first promoted to `__laccum`.

Because of the limited range of `__sfract` and `__fract`, only a few operations make sense when combining an integer with an `__sfract` or `__fract`. Therefore, the C compiler only supports the following operations for integers combined with fractional types:

left	operand	right	result
fractional	*	integer	fractional
integer	*	fractional	fractional
fractional	/	integer	fractional
fractional	<<	integer	fractional
fractional	>>	integer	fractional
fractional: __sfract, __fract			
integer: char, short, int, long, long long			

1.1.3. Packed Data Types

The TASKING C compiler additionally supports the packed types __packb and __packhw.

A __packb value consists of four signed or unsigned char values. A __packhw value consists of two signed or unsigned short values.

The TriCore instruction set supports a number of arithmetic operations on packed data types directly. For example, the following function:

```
__packb  add4 ( __packb a, __packb b )
{
    return a + b;
}
```

results into the following assembly code:

```
add4:
    add.b   d2,d4,d5
    ret16
```

Section 1.11.5, *Intrinsic Functions* explains intrinsic functions. Section 1.11.5.3, *Packed Data Type Support* lists the intrinsic functions.

Halfword packed unions and structures

To minimize space consumed by alignment padding with unions and structures, elements follow the minimum alignment requirements imposed by the architecture. The TriCore architecture supports access to 32-bit integer variables on halfword boundaries.

Because only doubles, circular buffers, __laccum or pointers require the full word access, structures that do not contain members of these types are automatically halfword (2 bytes) packed.

Structures and unions that are divisible by 64-bit or larger or contain members that are divisible by 64-bit or larger, are word packed to allow efficient access through LD.D and ST.D instructions. These load and store operations require word aligned structures that are divisible by 64-bit or larger. If necessary, 64-bit (or larger) divisible structure elements are aligned or padded to make the structure 64-bit accessible. If the LD.D/ST.D instructions are used depends on the tradeoff value (C compiler option **--tradeoff** (**-t**)): only for tradeoff values 0, 1 or 2 these instructions are used.

You can see the difference by using the following code (struct.c):

```
typedef struct
{
    char a;
    char b;
    char c;
    char d;

    char e;
    char f;
    char g;
```

```
        char h;

        char j;

} ST_64;

ST_64 st_64_1;
ST_64 st_64_2;

void main( void )
{
    st_64_1 = st_64_2;
}
```

and use the following invocations:

```
ctc struct.c -t0
ctc struct.c
```

With `#pragma pack 2` you can disable the `LD.D/ST.D` structure and union copy optimization to ensure halfword structure and union packing when possible. This "limited" halfword packing only supports structures and unions that do not contain double, circular buffer, `__laccum` or pointer type members and that are not qualified with `#pragma align` to get an alignment larger than 2-byte. With `#pragma pack 0` you turn off halfword packing again.

```
#pragma pack 2
typedef struct {
    unsigned char   uc1;
    unsigned int    ui1;
    unsigned short  us1;
    unsigned int    ui2;
    unsigned short  us2;
} packed_struct;
#pragma pack 0
```

When you place a `#pragma pack 0` before a structure or union, its alignment will not be changed:

```
#pragma pack 0
packed_struct pstruct;
```

The alignment of data sections and stack can also affect the alignment of the base address of a halfword packed structure. A halfword packed structure can be aligned on a halfword boundary or larger alignment. When located on the stack or at the beginning of a section, the alignment becomes a word, because of the minimum required alignment of data sections and stack objects. A stack or data section can contain any type of object. To avoid wrong word alignment of objects in the section, the section base is also word aligned.

1.1.4. Changing the Alignment: __align()

By default the TriCore compiler aligns objects to the minimum alignment required by the architecture. See Section 1.1, *Data Types*. With the attribute `__align()` you can change the alignment of objects of

a size greater than or equal to four bytes. The alignment must be a power of two and larger than or equal to 4.

> **Caution:** Use `__align()` with extreme care! You should know exactly how to use it properly, otherwise it may result in incorrect (trapping) code. You can only use it to increase the alignment; and the alignment must always be a multiple of the original required alignment!

> Do not use `__align()` on local variables. As local variables are placed on the stack or in registers the attribute `__align()` has no effect.

Example:

```
__align(4) int globalvar; /* changed to 4 bytes alignment
                             instead of default 2 bytes  */
```

Instead of the attribute `__align()` you can also use `__attribute__((__align(n)))` or `#pragma align`.

1.2. Accessing Memory

You can use static memory qualifiers to allocate static objects in a particular part of the addressing space of the processor.

In addition, you can place variables at absolute addresses with the keyword `__at()`.

1.2.1. Memory Qualifiers

In the C language you can specify that a variable must lie in a specific part of memory. You can do this with a *memory qualifier*.

You can specify the following memory qualifiers:

Qualifier	Description	Location	Maximum object size	Pointer size	Section types
`__near`[*]	Near data, direct addressable	First 16 kB of a 256 MB block	16 kB	32-bit	neardata, nearrom, nearbss, nearnoclear
`__far`[*]	Far data, indirect addressable	Anywhere	no limit	32-bit	fardata, farrom, farbss, farnoclear
`__a0`	Uninitialized / constant / cleared data	Sign-extended 16-bit offset from address register A0.	64 kB	32-bit	a0data, a0rom, a0bss
`__a1`	Uninitialized / constant / cleared data	Sign-extended 16-bit offset from address register A1.	64 kB	32-bit	a1data, a1rom, a1bss

Qualifier	Description	Location	Maximum object size	Pointer size	Section types
__a8	Uninitialized / constant / cleared data	Sign-extended 16-bit offset from address register A8.	64 kB	32-bit	a8data, a8rom, a8bss
__a9	Uninitialized / constant / cleared data	Sign-extended 16-bit offset from address register A9.	64 kB	32-bit	a9data, a9rom, a9bss

> * If you do not specify __near or __far, the compiler chooses where to place the declared object. With the C compiler option **--default-near-size** (maximum size in bytes for data elements that are by default located in __near sections) you can specify the size of data objects which the compiler then by default places in near memory.

All these memory qualifiers (__near, __far, __a0, __a1, __a8 and __a9) are related to the object being defined, they influence where the object will be located in memory. They are not part of the type of the object defined. Therefore, you cannot use these qualifiers in typedefs, type casts or for members of a struct or union.

Address registers A0, A1, A8, and A9 are designated as system global registers. They are not part of either context partition and are not saved/restored across calls. They can be protected against write access by user applications. A0, A1, A8 and A9 are freely to use on any type of data.

It is not allowed to assign ROM and RAM data to the same group. With the address registers A0, A1, A8 and A9 you can only access ROM or RAM but not both. Mixed ROM and RAM section types are not allowed for qualifiers __a0, __a1, __a8 and __a9. When this happens the linker issues the message:

```
Section .rodata_ax in group ax has memory type ROM but
expected memory type RAM like section .data_ax in the same group
```

where *x* is one of 0, 1 ,8, 9.

For example, it is not allowed to have variable a0rom and a0ram in a single application.

```
__a0 const int a0rom=1;
__a0       int a0ram=1; // Mixing ROM/RAM is not allowed
```

For this example the linker reports that the section .rodata_a0.a0.a0rom in group a0 has memory type ROM but expected memory type RAM like section .data_a0.a0.a0ram in the same group.

Examples using memory qualifiers

To declare a fast accessible integer in directly addressable memory:

```
int __near Var_in_near;
```

To allocate a pointer in far memory (the compiler will not use absolute addressing mode):

```
__far int * Ptr_in_far;
```

To declare and initialize a string in A0 memory:

```
char __a0 string[] = "TriCore";
```

If you use the __near memory qualifier, the compiler generates faster access code for those (frequently used) variables. Pointers are always 32-bit.

Functions are by default allocated in ROM. In this case you can omit the memory qualifier. You cannot use memory qualifiers for function return values.

Some examples of using pointers with memory qualifiers:

```
int __near * p;   /* pointer to int in __near memory
                     (pointer has 32-bit size)     */
int __far * g;    /* pointer to int in __far memory
                     (pointer has 32-bit size)     */

g = p;            /* the compiler issues a warning  */
```

You cannot use memory qualifiers in structure declarations:

```
struct S {
   __near int i;  /* put an integer in near
                      memory: Incorrect !      */
   __far int * p; /* put an integer pointer in
                      far memory: Incorrect !   */
   }
```

If a library function declares a variable in near memory and you try to redeclare the variable in far memory, the linker issues an error:

```
extern int __near foo;  /* extern int in near memory*/

int __far foo;          /* int in far memory         */
```

The usage of the variables is always without a storage specifier:

```
char __near example;
example = 2;
```

The generated assembly would be:

```
mov16 d15,2
st.b  example,d15
```

All allocations with the same storage specifiers are collected in units called 'sections'. The section with the __near attribute will be located within the first 16 kB of each 256 MB block.

1.2.2. Placing an Object at an Absolute Address: __at()

Just like you can declare a variable in a specific part of memory (using memory qualifiers), you can also place an object or a function at an absolute address in memory.

With the attribute `__at()` you can specify an absolute address. The address is a 32-bit linear address.

Examples

```
unsigned char Display[80*24] __at( 0x2000 );
```

The array `Display` is placed at address 0x2000. In the generated assembly, an absolute section is created. On this position space is reserved for the variable `Display`.

```
int i __at(0x1000) = 1;

void f(void) __at( 0xa0001000 ) { __nop(); }
```

The function `f` is placed at address 0xa0001000.

Restrictions

Take note of the following restrictions if you place a variable at an absolute address:

- The argument of the `__at()` attribute must be a constant address expression.

- You can place only global variables at absolute addresses. Parameters of functions, or automatic variables within functions cannot be placed at absolute addresses.

- A variable that is declared `extern`, is not allocated by the compiler in the current module. Hence it is not possible to use the keyword `__at()` on an external variable. Use `__at()` at the definition of the variable.

- You cannot place structure members at an absolute address.

- Absolute variables cannot overlap each other. If you declare two absolute variables at the same address, the assembler and/or linker issues an error. The compiler does not check this.

1.2.3. Accessing Bits

There are several methods to access single bits in a variable. The compiler generates efficient bit operations where possible.

Masking and shifting

The classic method to extract a single bit in C is masking and shifting.

```
unsigned int bitword;
void foo( void )
{
    if( bitword & 0x0004 )    // bit 2 set?
    {
        bitword &= ~0x0004;   // clear bit 2
    }
    bitword |= 0x0001;        // set bit 0;
}
```

Built-in macros __getbit() and __putbit()

The compiler has the built-in macros `__getbit()` and `__putbit()`. These macros expand to the `__extru()` and `__imaskldmst()` and intrinsic functions to perform the required result.

```
int * bw;
void foo( void )
{
    if( __getbit( bw, 2 ) )
    {
        __putbit( 0, bw, 2 );
    }
    __putbit( 1, bw, 0 );
}
```

Accessing bits using a struct/union combination

```
typedef union
{
    unsigned int word;
    struct
    {
        int  b0 : 1;
        int  b1 : 1;
        int  b2 : 1;
        int  b3 : 1;
        int  b4 : 1;
        int  b5 : 1;
        int  b6 : 1;
        int  b7 : 1;
        int  b8 : 1;
        int  b9 : 1;
        int  b10: 1;
        int  b11: 1;
        int  b12: 1;
        int  b13: 1;
        int  b14: 1;
        int  b15: 1;
    } bits;
} bitword_t;

bitword_t bw;

void foo( void )
{
    if( bw.bits.b3 )
    {
        bw.bits.b3 = 0;
    }
    bw.bits.b0 = 1;
```

```
}

void reset( void )
{
    bw.word = 0;
}
```

1.3. Data Type Qualifiers

1.3.1. Circular Buffers: __circ

The TriCore core has support for implementing specific DSP tasks, such as finite impulse response (FIR) and infinite impulse response (IIR) filters. For the FIR and IIR filters the TriCore architecture supports the circular addressing mode. The TriCore C compiler supports *circular buffers* for these DSP tasks. This way, the TriCore C compiler makes hardware features available at C source level instead of at assembly level only.

A *circular buffer* is a linear (one dimensional) array that you can access by moving a pointer through the data. The pointer can jump from the last location in the array to the first, or vice-versa (the pointer wraps-around). This way the buffer appears to be continuous. The TriCore C compiler supports the keyword __circ (circular addressing mode) for this type of buffer.

Example:

```
__fract __circ circbuf[10];
__fract __circ * ptr_to_circbuf = circbuf;
```

Here, circbuf is declared as a circular buffer. The compiler aligns the base address of the buffer on the access width (in this example an int, so 4 bytes). The compiler keeps the buffer size and uses it to control pointer arithmetic of pointers that are assigned to the buffer later.

Note that it is not allowed to declare an automatic circular buffer, because circular buffers require an alignment of 64-bit, but the TriCore stack uses a 32-bit alignment. Use keyword static for local circular buffers.

Circular pointers

You can perform operations on circular pointers with the usual C pointer arithmetic with the difference that the pointer will wrap. When you access the circular buffer with a circular pointer, it wraps at the buffer limits. Circular pointer variables are 64 bits in size.

Example:

```
while( *Pptr_to_circbuf++ );
```

Indexing

Indexing in the circular buffer, using an integer index, is treated equally to indexing in a non-circular array.

Example:

```
int i = circbuf[3];
```

The index is not calculated modulo; indexing outside the array boundaries yields undefined results.

Intrinsic function __initcirc()

If you want to initialize a circular pointer with a dynamically allocated buffer at run-time, you should use the intrinsic function __initcirc():

```
#define N 100
unsigned short s = sizeof(__fract);
__fract * buf = calloc( N, s );
__fract __circ * ptr_to_circbuf = __initcirc( buf, N * s, 0 * s );
```

1.3.2. Accessing Hardware from C

Using Special Function Registers

It is easy to access Special Function Registers (SFRs) that relate to peripherals from C. The SFRs are defined in a special function register file (*.sfr) as symbol names for use with the compiler. An SFR file contains the names of the SFRs and the bits in the SFRs.

Example use in C (SFRs from sfr/regtc1796b.sfr):

```
void set_sfr(void)
{
    SBCU_SRC.I |= 0xb32a;  /* access SBCU Service Request
                              Control register as a whole      */

    SBCU_SRC.B.SRE = 0x1;  /* access SRE bit-field of SBCU
                              Service Request Control register */
}
```

You can find a list of defined SFRs and defined bits by inspecting the SFR file for a specific processor. The files are located in the sfr subdirectory of the standard include directory. The files are named reg*cpu*.sfr, where *cpu* is the CPU specified with the control program option **--cpu**. The compiler includes this register file if you specify option **--include-file=sfr/regtc1796b.sfr**.

Defining Special Function Registers: __sfrbit16, __sfrbit32

SFRs are defined in SFR files and are written in C. With the data type qualifiers __sfrbit16 and __sfrbit32 you can declare bit-fields in special function registers.

According to the *TriCore Embedded Applications Binary Interface*, 'normal' bit-fields are accessed as char, short or int. Bit-fields are aligned according to the table in Section 1.1, *Data Types*.

If you declare bit-fields in special function registers, this behavior is not always desired: some special function registers require 16-bit or 32-bit access. To force 16-bit or 32-bit access, you can use the data type qualifiers __sfrbit16 and __sfrbit32.

When the SFR contains fields, the layout of the SFR is defined by a typedef-ed union. The following example is part of an SFR file and illustrates the declaration of a special function register using the data type qualifier `__sfrbit32`:

```
typedef volatile union
{
   struct
   {
   unsigned __sfrbit32 SRPN : 8;  /* Service Priority Number */
   unsigned __sfrbit32      : 2;
   unsigned __sfrbit32 TOS  : 2;  /* Type-of-Service Control */
   unsigned __sfrbit32 SRE  : 1;  /* Service Request Enable Control */
   unsigned __sfrbit32 SRR  : 1;  /* Service Request Flag */
   unsigned __sfrbit32 CLRR : 1;  /* Request Flag Clear Bit */
   unsigned __sfrbit32 SETR : 1;  /* Request Flag Set Bit */
   unsigned __sfrbit32      : 16;
   } B;

   int I;
   unsigned int U;
} LBCU_SRC_type;
```

Read-only fields can be marked by using the `const` keyword.

The SFR is defined by a cast to a 'typedef-ed union' pointer. The SFR address is given in parenthesis. Read-only SFRs are marked by using the `const` keyword in the macro definition.

```
#define LBCU_SRC (*(LBCU_SRC_type*)(0xF87FFEFCu))
                /* LBCU Service Control Register */
```

Restrictions

- You can use the `__sfrbit32` data type qualifier only on `int` bit-field types. The compiler issues an error if you use for example `__sfrbit32 char x : 8;`

- You can use the `__sfrbit16` data type qualifier only on `int` or `short` bit-field types. The compiler issues an error if you use for example `__sfrbit16 char x : 8;`

- When you use the `__sfrbit32` and `__sfrbit16` data type qualifiers on other types than a bit-field, the compiler ignores this without a warning. For example, `__sfrbit32 int global;` is equal to `int global;`.

- Structures or unions that contain a member qualified with `__sfrbit16`, are zero padded to complete a halfword if necessary. The structure or union will be halfword aligned. Structures or unions that contain a member qualified with `__sfrbit32`, are zero padded to complete a full word if necessary. The structure or union will be word aligned.

1.3.3. Saturation: __sat

When a variable is declared with the type qualifier `__sat`, all operations on that variable will be performed using saturating arithmetic. When an operation is performed on a plain variable and a `__sat` variable,

the __sat takes precedence, and the operation is done using saturating arithmetic. The type of the result of such an operation also includes the qualifier __sat, so that another operation on the result will also be saturated. In this respect, the behavior of the type qualifier __sat is comparable to the unsigned keyword. You can overrule this behavior by inserting type casts with or without the type qualifier __sat in an expression.

You can only use the type qualifier __sat on type int (fractional types are always saturated).

Care should be taken when combining signed and unsigned integers, since no saturation between signed and unsigned is done.

Examples:

```
__sat int si        = 0x7FFFFFFF;
 int i              = 0x12345;
 unsigned int ui    = 0xFFFFFFFF;


si + i  // a saturated addition is performed,
        // yielding a saturated int

si + ui // a saturated unsigned addition is performed,
        // yielding a saturated unsigned int

i + ui  // a normal unsigned addition is performed,
        // yielding an unsigned int
```

1.3.4. External MCS RAM Data References: __mcsram

You can reference external MCS RAM data from the TriCore with the keyword __mcsram. Only external variables can be qualified with the keyword __mcsram.

Global MCS RAM variables can only be defined in the MCS application. Only types with a size of 32-bit can have the keyword __mcsram, because only 32-bit types are supported by the MCS in the MCS RAM space.

To refer to a global variable *name* in a specific MCS core, you need to prefix the variable name with *core_*

For example, if in two MCS cores a global variable count is defined, you can reference them externally by the TriCore:

```
extern volatile int __mcsram mcs00_count; /* variable count in mcs00 */
extern volatile int __mcsram mcs01_count; /* variable count in mcs01 */
```

__mcsram external variables get the _lc_t_ linker prefix. _lc_t_mcs00_count and _lc_t_mcs01_count for the example above.

1.3.5. External PCP PRAM Data References: __pram

You can reference external PCP PRAM data from the TriCore with the keyword __pram. Only external variables can be qualified with the keyword __pram.

Global PRAM variables can only be defined in the PCP application. Only types with a size of 32-bit can have the keyword `__pram`, because only 32-bit types are supported by the PCP in the PRAM space.

For example, in the PCP source `channel3.c` of the `pcp-multi-ch3-ch4` example delivered with the product, the following variable is defined:

```
int channel3_count = 0;
```

In the TriCore source `tc_main.c` of the `pcp-multi-start` example delivered with the product, this global variable defined in a PCP channel is referenced externally by the TriCore. The symbol prefix used by the PCP for this channel is `_PCP_`.

```
extern volatile int __pram _PCP_channel3_count;
```

`__pram` external variables get the `_lc_s_` linker prefix.

When `__pram` is used for a TriCore 1.6.x or 1.6.2 core, a warning is generated and the keyword is ignored. The warning generated is:

```
W757: bad symbol attribute __pram for object "_PCP_channel3_count" -- ignored
```

1.3.6. Shared Data between TriCore and PCP: __share_pcp

Global data can be shared between the TriCore and PCP with the keyword `__share_pcp`. Only global and external variables can be qualified with the keyword `__share_pcp`. `__share_pcp` global variables get the `_lc_` linker prefix.

External reference of TriCore global variables from PCP

For example, in the TriCore source `tc_main.c` of the `pcp-multi-start` example delivered with the product, the following variable is defined:

```
volatile int __far __share_pcp shared_CPU_FPI;
```

In the PCP source `channel1.c` of the `pcp-multi-ch1` example delivered with the product, this variable is referenced as:

```
extern int __far __share_pcp shared_CPU_FPI;
```

When `__share_pcp` is used for a TriCore 1.6.x or 1.6.2 core, a warning is generated and the keyword is ignored. The warning generated is:

```
W757: bad symbol attribute __share_pcp for object "shared_CPU_FPI" -- ignored
```

External reference of PCP global variables from TriCore

For example, in the PCP source `channel1.c` of the `pcp-multi-ch1` example delivered with the product, the following variable is defined:

```
int __far __share_pcp channel2_shared_PCP_FPI = 0;
```

In the TriCore source `tc_main.c` of the `pcp-multi-start` example delivered with the product, this variable is referenced as:

```
extern volatile int __far __share_pcp channel2_shared_PCP_FPI;
```

External reference of PCP PRAM variables that have application scope

To externally reference PCP PRAM variables that have application scope instead of channels scope, you have to combine the keywords `__share_pcp` and `__pram`. Global PCP variables that are being shared between PCP channels do not use a PCP channel symbol prefix. `__pram __share_pcp` external variables get the `_lc_s__lc_` linker prefix.

For example, in the PCP source `channel1.c` of the `pcp-multi-ch1` example delivered with the product, the following variable is shared between different PCP channels:

```
int __share_pcp channel1_shared_PCP_PRAM = 0;
```

In the TriCore source `tc_main.c` of the `pcp-multi-start` example delivered with the product, this variable is referenced as:

```
extern volatile int __pram __share_pcp channel1_shared_PCP_PRAM;
```

1.4. Multi-Core Support

The TASKING VX-toolset for TriCore has support for multi-core versions of the TriCore. For example, the TC27x contains three TriCore cores (core 0, core 1 and core 2). Each core has local memory for data (data scratchpad RAM, DSPR*n*) and code (program scratchpad RAM, PSPR*n*). This local memory is mirrored and can also be accessed by the other cores. Code can be shared between binary compatible cores. Data can be shared between cores when the memory system provides access to the data. To define how code and data is accessible from one or multiple cores you can specify code core associations and data core associations, as explained in the following sections.

Note that accessing data or code scratch pad memory from another core (which is only possible via global addresses) may result in an extra instruction cycle penalty depending on read/write, DSPR/PSPR, while the core itself can access its own data or code scratchpad memory (via local and global addressing modes) without this penalty. The following table describes the CPU access times in CPU clock cycles. The access times are described as maximum CPU stall cycles where e.g. a data access to the local DSPR results in zero stall cycles. Note that the CPU does not always immediately stall after the start of a data read from another SPR due to instruction pipelining effects. This means that the average number will be lower than the shown numbers.

CPU Access Mode	CPU clock cycles	
	TC26x / TC27x	**TC29x**
Data read access to own DSPR	0	0
Data write access to own DSPR	0	0
Data read access to own or other PSPR	5	8
Data write access to own or other PSPR	0	0
Data read access to other DSPR	5	8
Data write access to other DSPR	0	0
Instruction fetch from own PSPR	0	0
Instruction fetch from other PSPR (critical word)	5	8
Instruction fetch from other PSPR (any remaining words)	0	0
Instruction fetch from other DSPR (critical word)	5	8
Instruction fetch from other DSPR (any remaining words)	0	0

Also note that multi-core is supported for the TriCore 1.6.x and 1.6.2 architectures only. This is automatically selected when you select, for example, the TC27x in Eclipse. If you build your sources on the command line with the control program, you have to specify control program option **--cpu=tc27x**. The control program will call the C compiler, assembler and linker with the correct options.

If you want to build an application for a single-core configuration, you also need to select the specific core in Eclipse (for example, TriCore core 0). If you build your sources on the command line with the control program, you also have to specify control program option **--lsl-core=tc0**.

Code and data can be shared, private or cloned.

Shared

In the default situation all code and data are accessible by all cores. The symbols are located in shared memory.

Private

Private means that the code and/or data is copied to, and accessed from, the local scratchpad memory of one particular core.

Cloned

Cloned means that code or data is copied to the local scratchpad memory of each binary compatible core, or a specific core. The core then treats the code/data as if it were private.

1.4.1. Data Core Association

The term "data core association" (DCA) is used to define:

• whether a data object is accessible from one or multiple cores

• the type of memory where the data will be allocated

• the number of memory instances of the data object

You can use a memory qualifier (`__share`, `__private0`, `__private1`, `__private2` or `__clone`) or pragma (`#pragma data_core_association`) to qualify individual data objects, or you can use an option (C compiler option **--data-core-association**) to specify the default data core association. The default data core association is "share". This means that all data can be accessed by all cores.

Data core association	Memory qualifier	Accessible from [*]	Number of instances	Allocation in
Share	`__share`	All cores	One instance. The data object is shared between cores.	Global RAM or core-local RAM Note: when allocated in core-local ram the shared object is accessed via the mirror page.

Private	`__private0`
	`__private1`
	`__private2`
		One core	One instance. For one specific core.	Core-local RAM

Clone	`__clone`	All cores	Multiple instances. For each core one instance is allocated. All instances will be located at identical addresses.	Core-local RAM

[*] Note that `__private`*n* data objects are private for the core from a compiler point of view. At run-time, the MPU (Memory Protection Unit) must be configured to protect memory from being accessed from other cores.

Instead of a memory qualifier you can also use a pragma:

```
#pragma data_core_association share | private{012} | clone
```

Based on the specified data core association the compiler stores the data object in a section with the following naming convention:

```
section_type_prefix.share|private{012}|clone.module_name.symbol_name
```

For more information on section names see Section 1.12, *Compiler Generated Sections*.

The linker recognizes the section names, duplicates clone sections for each binary compatible core and locates core specific code and data in the scratchpad memory of each core, resulting in one absolute object file (ELF) for each binary compatible set of cores.

1.4.2. Code Core Association

The term "code core association" (CCA) is used to define:

- the core or cores that are allowed to execute a function

- the type of memory where the function will be allocated

- the number of instances that are copied to local scratchpad RAM, i.e. the number of entries in the copy table

- a restriction on the type of data (defined by a data core association) the code may access

You can use a memory qualifier (`__share`, `__private0`, `__private1`, `__private2` or `__clone`) or pragma (`#pragma code_core_association`) to qualify individual functions, or you can use an option (C compiler option **--code-core-association**) to change the default code core association. The default code core association is "share". This means that all code can be executed by all cores.

Code core association	Memory qualifier	Executes on [*]	Number of instances	Allowed access of DCA qualified data	Allocation in
Share	`__share`	Any core	One instance. The code is shared between cores.	Shared Cloned (of executing core)	PFLASH_0, PFLASH_1 or core-local RAM
Private	`__private0` `__private1` `__private2`	One core only	One instance.	Shared Private Cloned	Core-local RAM

Code core association	Memory qualifier	Executes on *	Number of instances	Allowed access of DCA qualified data	Allocation in
Clone	`__clone`	Any core	Multiple instances. Each code instance is executed by one core.	Shared Cloned	Core-local RAM

* Note that `__private`*n* code is private for the core from a compiler point of view. At run-time, the MPU (Memory Protection Unit) must be configured to protect code on one core from being executed by other cores.

Instead of a memory qualifier you can also use a pragma:

```
#pragma code_core_association share | private{012} | clone
```

Based on the specified code core association the compiler stores the code object in a section with the following naming convention:

section_type_prefix`.share|private{012}|clone.`*module_name.symbol_name*

For more information on section names see Section 1.12, *Compiler Generated Sections*.

The linker recognizes the section names, duplicates clone sections for each binary compatible core and locates core specific code and data in the scratchpad memory of each core, resulting in one absolute object file (ELF) for each binary compatible set of cores.

1.4.3. Core Association Restrictions

The following restriction apply to core associations:

Run-time bounds data is shared by all cores

For run-time bounds checking, bounds data is generated in a section declared with a fixed section name 'bounds'. No data core association is applied to this section. The linker uses the default core association share. The bounds data is shared among all cores.

Example (`bounds.c`):

```
typedef struct
{
    int i;
} s_t;

s_t s;
const int ci1  =55;
const int ci2  =55;

int main(void)
{
```

```
    s.i = 42;

    return ci1+ci2;
}
```

Use the following command to see the result:

```
ctc bounds.c --runtime --core=tc1.6.x --data-core-association=share
    --code-core-association=share
```

Profile data is shared by all cores

For dynamic profiling, profiling data is generated in sections declared with fixed section names '*__prof_'. No data core association is applied to these sections. The linker uses the default core association share. The profiling data is shared among all cores.

Example (prof.c):

```
void f1(int i);
void f2(int i);
void f3(int i);

void f3(int i)
{
    if ( i )
    {
        f1(i-1);
        f2(i-1);
        f3(i-1);
    }
}

void f2(int i)
{
    if ( i )
    {
        f1(i-1);
        f2(i-1);
        f3(i-1);
    }
}

void f1(int i)
{
    if ( i )
    {
        f1(i-1);
        f2(i-1);
        f3(i-1);
    }
}
```

```
void main(void)
{
    f1(3);
    f2(3);
    f3(3);
}
```

When you compile with:

```
ctc prof.c --profile --core=tc1.6.x --data-core-association=share
    --code-core-association=share
```

This results in the following section declarations:

```
.sdecl '.zbss.prof._999001___prof_counter_0',data
.sdecl '.zbss.prof._999002___prof_counter_0',data
.sdecl '.zbss.prof._999003___prof_counter_0',data
.sdecl '.zbss.prof._999004___prof_counter_0',data
```

Predefined identifier __func__ is shared by all cores

You cannot use the data core association symbol qualifiers or pragmas to associate a core with predefined identifier `__func__`. The linker uses the default core association share. `__func__` is shared among all cores.

Example (`func.c`):

```
char funcname[10];

void function( void )
{
    for(int i = 0; i < 6; i++ ) {
        funcname[i] = __func__[i];
    }
}
```

When you compile with:

```
ctc func.c --core=tc1.6.x --data-core-association=share
    --code-core-association=share
```

This results in the following section declaration:

```
.sdecl '.rodata.func._999001___func__',data,rom
```

No core is associated to a section renamed with the attribute section

No section prefixing is supported on sections that are renamed with attribute `section`. The linker uses the default core association share. Of course it is still possible to use the core association section naming convention in the section attribute to do the core association manually.

Example (`section.c`):

```
int function( void ) __section__("fixed_section_name")
{
    return 0;
}
```

When you compile with:

```
ctc section.c --core=tc1.6.x --data-core-association=share
    --code-core-association=share
```

This results in the following section declaration:

```
  .sdecl 'fixed_section_name',code
```

Renaming with `#pragma section` supports the normal section prefixing.

```
#pragma section code "myname"

int function( void )
{
    return 0;
}
```

results in the following section declaration:

```
  .sdecl '.text.share.myname',code
```

For more details see Section 1.12, *Compiler Generated Sections*.

1.4.4. Core Association and Addressing Modes

You can combine the data core associations with the memory qualifiers from Section 1.2.1, *Memory Qualifiers*. The most efficient way is to qualify cloned and private data objects as `__near`. You can use the qualifier `__near` explicitly or you can use the C compiler option **--default-near-size=***value*. All data objects with a size less than or equal to *value* are located in `__near` sections.

Data objects located in scratchpad RAM of core *N* can be treated as `__near` by core *N*. Other cores need to access the data object through the mirror pages and you have to use `__far` addressing. This results into the following scheme where `__near` means that `__near` access *may* be used, and `__far` means that `__far` access must be used:

Code core association	Data core association	Applicable addressing mode(s)
Share	Share	__far, __a[0|1|8|9]
	Private	__near
	Clone	__near
Private (RAM)	Share	__far, __a[0|1|8|9]

Code core association	Data core association	Applicable addressing mode(s)
	Private	__near
	Clone	__near
Clone	Share	__far, __a[0|1|8|9]
	Clone	__near

As a consequence shared data located in scratchpad memory of core *N* is accessed via __*far* addressing, also by the code executing on core *N*.

1.4.5. Core Association and Function Calls

The code core association affects caller-callee relations. Private functions are not accessible by each core. Therefore, calling a private function is illegal unless it is guaranteed that the code that contains the call can only be executed by the core associated with the private function.

Both the C compiler and linker check for illegal function calls. However, the C compiler and linker cannot check indirect calls and the C compiler cannot check calls to external functions, due to lack of type information.

The following table shows the relation between function calls and code core associations.

Code core association of caller	Code core association of callee	Issues
Share	Share	No issues.
	Private	Illegal call. A shared function is not allowed to call a private function.
	Clone	No issues.
Private	Share	No issues.
	Private	Caller and callee must be associated with the same core.
	Clone	No issues.
Clone	Share	No issues.
	Private	Illegal call. A cloned function is not allowed to call a private function.
	Clone	No issues.

1.5. Shift JIS Kanji Support

In order to allow for Japanese character support on non-Japanese systems (like PCs), you can use the Shift JIS Kanji Code standard. This standard combines two successive ASCII characters to represent one Kanji character. A valid Kanji combination is only possible within the following ranges:

- First (high) byte is in the range 0x81-0x9f or 0xe0-0xef.

• Second (low) byte is in the range 0x40-0x7e or 0x80-0xfc

Compiler option **-Ak** enables support for Shift JIS encoded Kanji multi-byte characters in strings and (wide) character constants. Without this option, encodings with 0x5c as the second byte conflict with the use of the backslash ('\') as an escape character. Shift JIS in comments is supported regardless of this option.

Note that Shift JIS also includes Katakana and Hiragana.

Example:

```
// Example usage of Shift JIS Kanji
// Do not switch off option -Ak
// At the position of the italic text you can
// put your Shift JIS Kanji code
int  i;  // put Shift JIS Kanji here
char c1;
char c2;
unsigned int ui;
const char mes[]="put Shift JIS Kanji here";
const unsigned int ar[5]={'K','a','n',
                   'j','i'};
                   // 5 Japanese array
void main(void)
{
    i=(int)c1;
    i++; /* put Shift JIS Kanji here\
            continuous comment */
    c2=mes[9];
    ui=ar[0];
}
```

1.6. Using Assembly in the C Source: __asm()

With the keyword `__asm()` you can use assembly instructions in the C source and pass C variables as operands to the assembly code.

It is recommended to use constructs in C or use intrinsic functions instead of `__asm()`. Be aware that C modules that contain assembly are not portable and harder to compile in other environments.

The compiler does not interpret assembly blocks but passes the assembly code to the assembly source file; they are regarded as a black box. So, it is your responsibility to make sure that the assembly block is syntactically correct. Possible errors can only be detected by the assembler.

You need to tell the compiler exactly what happens in the inline assembly code because it uses that for code generation and optimization. The compiler needs to know exactly which registers are written and which registers are only read. For example, if the inline assembly writes to a register from which the compiler assumes that it is only read, the generated code after the inline assembly is based on the fact that the register still contains the same value as before the inline assembly. If that is not the case the

results may be unexpected. Also, an inline assembly statement using multiple input parameters may be assigned the same register if the compiler finds that the input parameters contain the same value. As long as this register is only read this is not a problem.

General syntax of the __asm keyword

```
__asm( "instruction_template"
        [ : output_param_list
        [ : input_param_list
        [ : register_reserve_list]]] );
```

instruction_template	Assembly instructions that may contain parameters from the input list or output list in the form: **%***parm_nr*[**.***regnum*]
%*parm_nr*[**.***regnum*]	Parameter number in the range 0 .. 9. With the optional **.***regnum* you can access an individual register from a register pair. For example, with register pair d0/d1, .0 selects register d0.
output_param_list	[[**"=**[**&**]*constraint_char***"(***C_expression***)**],...]
input_param_list	[[**"***constraint_char***"(***C_expression***)**],...]
&	Says that an output operand is written to before the inputs are read, so this output must not be the same register as any input.
constraint _char	Constraint character: the type of register to be used for the *C_expression*. See the table below.
C_expression	Any C expression. For output parameters it must be an lvalue, that is, something that is legal to have on the left side of an assignment.
register_reserve_list	[["*register_name*"],...]
register_name	Name of the register you want to reserve. For example because this register gets clobbered by the assembly code. The compiler will not use this register for inputs or outputs. Note that reserving too many registers can make register allocation impossible.

Specifying registers for C variables

With a *constraint character* you specify the register type for a parameter.

You can reserve the registers that are used in the assembly instructions, either in the parameter lists or in the reserved register list (*register_reserve_list*). The compiler takes account of these lists, so no unnecessary register saves and restores are placed around the inline assembly instructions.

Constraint character	Type	Operand	Remark
a	address register	a0 .. a15	
b	address register pair	b2, b4, b6, b12, b14	b2 = pair a2/a3, b4 = a4/a5, ...
d	data register	d0 .. d15	
e	date register pair	e0, e2, ..., e14	e0 = pair d0/d1, e2 = d2/d3, ...
m	memory	*variable*	memory operand

Constraint character	Type	Operand	Remark
i	immediate value	*value*	
number	type of operand it is associated with	same as **%***number*	Input constraint only. The *number* must refer to an output parameter. Indicates that **%***number* and *number* are the same register.

> If an input parameter is modified by the inline assembly then this input parameter must also be added to the list of output parameters (see Example 6). If this is not the case, the resulting code may behave differently than expected since the compiler assumes that an input parameter is not being changed by the inline assembly.

Loops and conditional jumps

The compiler does not detect loops with multiple __asm() statements or (conditional) jumps across __asm() statements and will generate incorrect code for the registers involved.

If you want to create a loop with __asm(), the whole loop must be contained in a single __asm() statement. The same counts for (conditional) jumps. As a rule of thumb, all references to a label in an __asm() statement must be in that same statement. You can use numeric labels for these purposes.

Example 1: no input or output

A simple example without input or output parameters. You can use any instruction or label. When it is required that a sequence of __asm() statements generates a contiguous sequence of instructions, then they can be best combined to a single __asm() statement. Compiler optimizations can insert instruction(s) in between __asm() statements. Use newline characters '\n' to continue on a new line in a __asm() statement. For multi-line output, use tab characters '\t' to indent instructions.

```
__asm( "nop\n"
       "\tnop" );
```

Example 2: using output parameters

Assign the result of inline assembly to a variable. With the constraint m memory is chosen for the parameter; the compiler decides where to put the variable. The %0 in the instruction template is replaced with the name of the variable.

```
__near int out;
void func( void )
{
   __asm( "mov    d15,#1234\n"
          "\tst.w  %0,d15"
          : "=m" (out) );
}
```

Generated assembly code:

```
mov   d15,#1234
st.w  out,d15
```

Example 3: using input parameters

Assign a variable to a memory location. A data register is chosen for the parameter because of the constraint d; the compiler decides which register is best to use. The %0 in the instruction template is replaced with the name of this register. The compiler generates code to move the input variable to the input register. Because there are no output parameters, the output parameter list is empty. Only the colon has to be present.

```
int in;
void initmem( void )
{
    __asm( "ST.W  0xa0000000,%0"
           :
           : "d" (in) );
}
```

Generated assembly code:

```
ld.w  d15,in
ST.W  0xa0000000,d15
```

Example 4: using input and output parameters

Multiply two C variables and assign the result to a third C variable. Data type registers are necessary for the input and output parameters (constraint d, %0 for out, %1 for in1, %2 for in2 in the instruction template). The compiler generates code to move the input expressions into the input registers and to assign the result to the output variable.

```
int in1, in2, out;

void multiply32( void )
{
  __asm( "mul %0, %1, %2"
         : "=d" (out)
         : "d" (in1), "d" (in2) );
}
```

Generated assembly code:

```
ld.w   d15,in1
ld.w   d0,in2
mul    d15, d15, d0
st.w   out,d15
```

Example 5: reserving registers

Sometimes an instruction knocks out certain specific registers. The most common example of this is a function call, where the called function is allowed to do whatever it likes with some registers. If this is the case, you can list specific registers that get clobbered by an operation after the inputs.

Same as *Example 4*, but now register `d0` is a reserved register. You can do this by adding a reserved register list (`: "d0"`). As you can see in the generated assembly code, register `d0` is not used (the first register used is `d1`).

```
int in1, in2, out;

void multiply32( void )
{
   __asm( "mul %0, %1, %2"
          : "=d" (out)
          : "d" (in1), "d" (in2)
          : "d0" );
}
```

Generated assembly code:

```
    ld.w    d15,in1
    ld.w    d1,in2
    mul     d15, d15, d1
    st.w    out,d15
```

Example 6: use the same register for input and output

As input constraint you can use a number to refer to an output parameter. This tells the compiler that the same register can be used for the input and output parameter. When the input and output parameter are the same C expression, these will effectively be treated as if the input parameter is also used as output. In that case it is allowed to write to this register. For example:

```
inline int foo(int par1, int par2, int * par3)
{
  int retvalue;

  __asm(
    "sh      %1,#-2\n\t"
    "add     %2,%1\n\t"
    "st.w    [%5],%2\n\t"
    "mov     %0,%2"
    : "=&d" (retvalue), "=d" (par1), "=d" (par2)
    : "1" (par1), "2" (par2), "a" (par3)
  );
  return retvalue;
}

int result,parm;
```

```
void func(void)
{
  result = foo(1000,1000,&parm);
}
```

In this example the "1" constraint for the input parameter `par1` refers to the output parameter `par1`, and similar for the "2" constraint and `par2`. In the inline assembly `%1` (`par1`) and `%2` (`par2`) are written. This is allowed because the compiler is aware of this.

This results in the following generated assembly code:

```
mov   d15,#1000
lea   a15,parm
mov   d0,d15

sh    d15,#-2
add   d0,d15
st.w  [a15],d0
mov   d1,d0

st.w  result,d1
```

However, when the inline assembly would have been as given below, the compiler would have assumed that `%1` (`par1`) and `%2` (`par2`) were read-only. Because of the `inline` keyword the compiler knows that `par1` and `par2` both contain 1000. Therefore the compiler can optimize and assign the same register to `%1` and `%2`. This would have given an unexpected result.

```
__asm(
    "sh      %1,#-2\n\t"
    "add     %2,%1\n\t"
    "st.w    [%3],%2\n\t"
    "mov     %0,%2"
    : "=&d" (retvalue)
    : "d" (par1), "d" (par2), "a" (par3)
);
```

Generated assembly code:

```
mov   d15,#1000
lea   a15,parm

sh    d15,#-2
add   d15,d15      ; same register, but is expected read-only
st.w  [a15],d15
mov   d0,d15

st.w  result,d0    ; contains unexpected result
```

Example 7: accessing individual registers in a register pair

You can access the individual registers in a register pair by adding a '.' after the operand specifier in the assembly part, followed by the index in the register pair.

```
int out1, out2;

void foo(double din)
{
    __asm ("ld.w %0, %2.0\n"
        "\tld.w %1, %2.1":"=&d"(out1),"=d"(out2):"e"(din) );
}
```

The first `ld.w` instruction uses index #0 of argument 2 (which is a double placed in a DxDx register) and the second `ld.w` instruction uses index #1. The input operand is located in register pair d4/d5. The assembly output becomes:

```
    ld.w    d15, d4
    ld.w    d0, e4,1 ; note that e4,1 corresponds to d5
    st.w    out1,d15
    st.w    out2,d0
```

If the index is not a valid index (for example, the register is not a register pair, or the argument has not a register constraint), the '.' is passed into the assembly output. This way you can still use the '.' in assembly instructions.

1.7. Attributes

You can use the keyword `__attribute__` to specify special attributes on declarations of variables, functions, types, and fields.

Syntax:

```
__attribute__((name,...))
```

or:

```
__name__
```

The second syntax allows you to use attributes in header files without being concerned about a possible macro of the same name. This second syntax is only possible on attributes that do not already start with an underscore. For example, you may use `__noreturn__` instead of `__attribute__((noreturn))`.

alias("*symbol*")

You can use `__attribute__((alias("symbol")))` to specify that the function declaration appears in the object file as an alias for another symbol. For example:

```
void __f() { /* function body */; }
void f() __attribute__((weak, alias("__f")));
```

declares 'f' to be a weak alias for '__f'.

__align(*value*)

You can use `__attribute__((__align(n)))` to change the alignment of objects of a size greater than or equal to four bytes. The alignment must be a power of two and larger than or equal to 4. See also Section 1.1.4, *Changing the Alignment: __align()*.

const

You can use `__attribute__((const))` to specify that a function has no side effects and will not access global data. This can help the compiler to optimize code. See also attribute `pure`.

The following kinds of functions should not be declared `__const__`:

• A function with pointer arguments which examines the data pointed to.

• A function that calls a non-const function.

export

You can use `__attribute__((export))` to specify that a variable/function has external linkage and should not be removed. During MIL linking, the compiler treats external definitions at file scope as if they were declared `static`. As a result, unused variables/functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module. During MIL linking not all uses of a variable/function can be known to the compiler. For example when a variable is referenced in an assembly file or a (third-party) library. With the `export` attribute the compiler will not perform optimizations that affect the unknown code.

```
int i __attribute__((export)); /* 'i' has external linkage */
```

flatten

You can use `__attribute__((flatten))` to force inlining of all function calls in a function, including nested function calls.

Unless inlining is impossible or disabled by `__attribute__((noinline))` for one of the calls, the generated code for the function will not contain any function calls.

format(*type*,*arg_string_index*,*arg_check_start*)

You can use `__attribute__((format(type,arg_string_index,arg_check_start)))` to specify that functions take `printf`, `scanf`, `strftime` or `strfmon` style arguments and that calls to these functions must be type-checked against the corresponding format string specification.

type determines how the format string is interpreted, and should be `printf`, `scanf`, `strftime` or `strfmon`.

arg_string_index is a constant integral expression that specifies which argument in the declaration of the user function is the format string argument.

arg_check_start is a constant integral expression that specifies the first argument to check against the format string. If there are no arguments to check against the format string (that is, diagnostics should only be performed on the format string syntax and semantics), *arg_check_start* should have a value of 0. For `strftime`-style formats, *arg_check_start* must be 0.

Example:

```
int foo(int i, const char * my_format, ...) __attribute__((format(printf, 2, 3)));
```

The format string is the second argument of the function `foo` and the arguments to check start with the third argument.

jump

You can use `__attribute__((jump))` to specify that a function can only be jumped to. The compiler generates a jump instruction instead of a call instruction. This is for example used in the startup code generation:

```
static void __noinline__ __noreturn__ __jump__ _start( void );
```

When you call `_start()` in your C source, the compiler generates:

```
j _start
```

leaf

You can use `__attribute__((leaf))` to specify that a function is a leaf function. A leaf function is an external function that does not call a function in the current compilation unit, directly or indirectly. The attribute is intended for library functions to improve dataflow analysis. The attribute has no effect on functions defined within the current compilation unit.

malloc

You can use `__attribute__((malloc))` to improve optimization and error checking by telling the compiler that:

• The return value of a call to such a function points to a memory location or can be a null pointer.

• On return of such a call (before the return value is assigned to another variable in the caller), the memory location mentioned above can be referenced only through the function return value; e.g., if the pointer value is saved into another global variable in the call, the function is not qualified for the malloc attribute.

• The lifetime of the memory location returned by such a function is defined as the period of program execution between a) the point at which the call returns and b) the point at which the memory pointer is passed to the corresponding deallocation function. Within the lifetime of the memory object, no other calls to malloc routines should return the address of the same object or any address pointing into that object.

noinline

You can use `__attribute__((noinline))` to prevent a function from being considered for inlining. Same as keyword `__noinline` or `#pragma noinline`.

always_inline

With `__attribute__((always_inline))` you force the compiler to inline the specified function, regardless of the optimization strategy of the compiler itself. Same as keyword `inline` or `#pragma inline`.

noreturn

Some standard C function, such as abort and exit cannot return. The C compiler knows this automatically. You can use `__attribute__((noreturn))` to tell the compiler that a function never returns. For example:

```
void fatal() __attribute__((noreturn));

void fatal( /* ... */ )
{
  /* Print error message */
  exit(1);
}
```

The function `fatal` cannot return. The compiler can optimize without regard to what would happen if `fatal` ever did return. This can produce slightly better code and it helps to avoid warnings of uninitialized variables.

protect

You can use `__attribute__((protect))` to exclude a variable/function from the duplicate/unreferenced section removal optimization in the linker. When you use this attribute, the compiler will add the "protect" section attribute to the symbol's section. Example:

```
int i __attribute__((protect));
```

Note that the protect attribute will not prevent the compiler from removing an unused variable/function (see the `used` symbol attribute).

This attribute is the same as `#pragma protect/endprotect`.

pure

You can use `__attribute__((pure))` to specify that a function has no side effects, although it may read global data. Such pure functions can be subject to common subexpression elimination and loop optimization. See also attribute `const`.

section("*section_name*")

You can use `__attribute__((section("name")))` to specify that a function must appear in the object file in a particular section. For example:

```
extern void foobar(void) __attribute__((section("bar")));
```

puts the function `foobar` in the section named `bar`.

> Note that this a GNU style attribute. It does not follow the TriCore EABI guidelines. It does not add the section prefix as with `#pragma section`. It gives you full control over the section name. So, to be EABI compliant make sure you provide the correct section prefix.

See also `#pragma section` and Section 1.12, *Compiler Generated Sections*.

used

You can use `__attribute__((used))` to prevent an unused symbol from being removed, by both the compiler and the linker. Example:

```
static const char copyright[] __attribute__((used)) = "Copyright 2015 Altium BV";
```

When there is no C code referring to the `copyright` variable, the compiler will normally remove it. The `__attribute__((used))` symbol attribute prevents this. Because the linker should also not remove this symbol, `__attribute__((used))` implies `__attribute__((protect))`.

unused

You can use `__attribute__((unused))` to specify that a variable or function is possibly unused. The compiler will not issue warning messages about unused variables or functions.

weak

You can use `__attribute__((weak))` to specify that the symbol resulting from the function declaration or variable must appear in the object file as a weak symbol, rather than a global one. This is primarily useful when you are writing library functions which can be overwritten in user code without causing duplicate name errors.

See also `#pragma weak`.

1.8. Pragmas to Control the Compiler

Pragmas are keywords in the C source that control the behavior of the compiler. Pragmas overrule compiler options. Put pragmas in your C source where you want them to take effect. Unless stated otherwise, a pragma is in effect from the point where it is included to the end of the compilation unit or until another pragma changes its status.

The syntax is:

```
#pragma [label:]pragma-spec pragma-arguments [on | off | default | restore]
```

or:

```
_Pragma( "[label:]pragma-spec pragma-arguments [on | off | default | restore]" )
```

Some pragmas can accept the following special arguments:

`on`	switch the flag on (same as without argument)
`off`	switch the flag off
`default`	set the pragma to the initial value
`restore`	restore the previous value of the pragma

Label pragmas

Some pragmas support a label prefix of the form "*label*:" between `#pragma` and the pragma name. Such a label prefix limits the effect of the pragma to the statement following a label with the specified name. The `restore` argument on a pragma with a label prefix has a special meaning: it removes the most recent definition of the pragma for that label.

You can see a label pragma as a kind of macro mechanism that inserts a pragma in front of the statement after the label, and that adds a corresponding `#pragma ... restore` after the statement.

Compared to regular pragmas, label pragmas offer the following advantages:

• The pragma text does not clutter the code, it can be defined anywhere before a function, or even in a header file. So, the pragma setting and the source code are uncoupled. When you use different header files, you can experiment with a different set of pragmas without altering the source code.

• The pragma has an implicit end: the end of the statement (can be a loop) or block. So, no need for pragma restore / endoptimize etc.

Example:

```
#pragma lab1:optimize P

volatile int v;

void f( void )
{
    int i, a;
```

```
      a = 42;

lab1: for( i=1; i<10; i++ )
      {
        /* the entire for loop is part of the pragma optimize */
        a += i;
      }
      v = a;
}
```

Supported pragmas

The compiler recognizes the following pragmas, other pragmas are ignored. On the command line you can use **ctc --help=pragmas** to get a list of all supported pragmas. Pragmas marked with (*) support a label prefix.

STDC FP_CONTRACT [on | off | default | restore] (*)

This is a standard ISO C99 pragma. With this pragma you can control the **+contract** flag of C compiler option **--fp-model**.

alias *symbol=defined_symbol*

Define *symbol* as an alias for *defined_symbol*. It corresponds to a `.ALIAS` directive at assembly level. The *symbol* should not be defined elsewhere, and *defined_symbol* should be defined with static storage duration (not extern or automatic).

align {*value* | default | restore} (*)

Increase the alignment of objects of four bytes or longer. The alignment value must be a power of two and larger than or equal to 4. Value 0 defaults to the compiler natural object alignment.

See C compiler option **--align**.

clear / noclear [on | off | default | restore] (*)

By default, uninitialized global or static variables are cleared to zero on startup. With pragma `noclear`, this step is skipped. Pragma `clear` resumes normal behavior. This pragma applies to constant data as well as non-constant data.

See C compiler option **--no-clear**.

code_core_association {*value* | default | restore} (*)

Switch to another code core association, where *value* is one of share, private0, private1, private2 or clone. The code core association of this pragma is assigned to the functions declarations or definitions that follow.

See C compiler option **--code-core-association**.

This pragma is only supported for multi-core TriCore architectures (**--core=tricore1.6.x** or **--core=tricore1.6.2**).

compactmaxmatch {*value* | default | restore} (*)

With this pragma you can control the maximum size of a match.

See C compiler option **--compact-max-size**.

CPU_TC*num* [on | off | default | restore]   (*)

Use software workarounds for the specified functional problem. For the list of functional problem numbers see Chapter 18, *CPU Problem Bypasses and Checks*.

See also C compiler option **--silicon-bug**.

For example, to enable workarounds for problem CPU_TC.013, specify the following pragma (without the dot):

```
#pragma CPU_TC013
```

data_core_association {*value* | default | restore} (*)

Switch to another data core association, where *value* is one of share, private0, private1, private2 or clone. The data core association of this pragma is assigned to the data declarations that follow.

See C compiler option **--data-core-association**.

This pragma is only supported for multi-core TriCore architectures (**--core=tricore1.6.x** or **--core=tricore1.6.2**).

default_a0_size [*value*] [default | restore] (*)

With this pragma you can specify a threshold value for __a0 allocation.

See C compiler option **--default-a0-size (-Z)**.

default_a1_size [*value*] [default | restore] (*)

With this pragma you can specify a threshold value for __a1 allocation.

See C compiler option **--default-a1-size (-Y)**.

default_near_size [*value*] [default | restore] (*)

With this pragma you can specify a threshold value for __near allocation.

See C compiler option **--default-near-size (-N)**.

extension isuffix [on | off | default | restore] (*)

Enables a language extension to specify imaginary floating-point constants. With this extension, you can use an "i" suffix on a floating-point constant, to make the type `_Imaginary`.

```
float 0.5i
```

extern *symbol*

Normally, when you use the C keyword `extern`, the compiler generates an `.EXTERN` directive in the generated assembly source. However, if the compiler does not find any references to the `extern` symbol in the C module, it optimizes the assembly source by leaving the `.EXTERN` directive out.

With this pragma you can force an external reference (`.EXTERN` assembler directive), even when the *symbol* is not used in the module.

for_constant_data_use_memory *memory*
for_extern_data_use_memory *memory*
for_initialized_data_use_memory *memory*
for_uninitialized_data_use_memory *memory*

Use the specified memory for the type of data mentioned in the pragma name. You can specify the following memories: `near`, `far`, `a0`, `a1`, `a8` or `a9`.

This pragma overrules the pragmas `default_a0_size`, `default_a1_size`, `default_near_size`, and the memory qualifiers `__near` and `__far`.

fp_negzero [on | off | default | restore] (*)

With this pragma you can control the **+negzero** flag of C compiler option **--fp-model**.

fp_nonan [on | off | default | restore] (*)

With this pragma you can control the **+nonan** flag of C compiler option **--fp-model**.

fp_rewrite [on | off | default | restore] (*)

With this pragma you can control the **+rewrite** flag of C compiler option **--fp-model**.

immediate_in_code [on | off | default | restore] (*)

With this pragma you force the compiler to encode all immediate values into instructions.

See C compiler option **--immediate-in-code**.

indirect [on | off | default | restore] (*)

Generates code for indirect function calling.

See C compiler option **--indirect**.

indirect_runtime [on | off | default | restore] (*)

Generates code for indirect calls to run-time functions.

See C compiler option **--indirect-runtime**.

inline / noinline / smartinline [default | restore] (*)

See Section 1.11.3, *Inlining Functions: inline*.

inline_max_incr {*value* | default | restore} (*)
inline_max_size {*value* | default | restore} (*)

With these pragmas you can control the automatic function inlining optimization process of the compiler. It has effect only when you have enable the inlining optimization (C compiler option **--optimize=+inline**).

See C compiler options **--inline-max-incr / --inline-max-size**.

loop_alignment {*value* | default | restore} (*)

Specify the alignment loop bodies will get when **--loop=+value** is enabled. Loops are only aligned if the align-loop optimization is enabled and the tradeoff is set to speed (<=2).

See C compiler option **--loop-alignment**.

macro / nomacro [on | off | default | restore] (*)

Turns macro expansion on or off. By default, macro expansion is enabled.

maxcalldepth {*value* | default | restore} (*)

With this pragma you can control the maximum call depth. Default is infinite (-1).

See C compiler option **--max-call-depth**.

message "*message*" ...

Print the message string(s) on standard output.

nomisrac [*nr*,...] [default | restore] (*)

Without arguments, this pragma disables MISRA C checking. Alternatively, you can specify a comma-separated list of MISRA C rules to disable.

See C compiler option **--misrac** and Section 4.7.2, *C Code Checking: MISRA C*.

object_comment "*string*" ... | default | restore (*)

This pragma generates a `.comment` section in the assembly file with the specified string. After assembling, this string appears in the generated `.o` or `.elf` object file. If you specify this pragma more than once in the same module, only the last pragma has effect.

See C compiler option **--object-comment**.

optimize [*flags*] / endoptimize [default | restore] (*)

You can overrule the C compiler option **--optimize** for the code between the pragmas `optimize` and `endoptimize`. The pragma works the same as C compiler option **--optimize**.

See Section 4.6, *Compiler Optimizations*.

pack {0 | 2 | default | restore} (*)

Specifies packing of structures. See Section 1.1.3, *Packed Data Types*.

profile [*flags*] / endprofile [default | restore] (*)

Control the profile settings. The pragma works the same as C compiler option **--profile**. Note that this pragma will only be checked at the start of a function. `endprofile` switches back to the previous profiling settings.

profiling [on | off | default | restore] (*)

If profiling is enabled on the command line (C compiler option **--profile**), you can disable part of your source code for profiling with the pragmas `profiling off` and `profiling`.

protect / endprotect [on | off | default | restore] (*)

With these pragmas you can protect sections against linker optimizations. This excludes a section from unreferenced section removal and duplicate section removal by the linker. `endprotect` restores the default section protection.

runtime [*flags* | default | restore] (*)

With this pragma you can control the generation of additional code to check for a number of errors at run-time. The pragma argument syntax is the same as for the arguments of the C compiler option **--runtime**. You can use this pragma to control the run-time checks for individual statements. In addition, objects declared when the "bounds" sub-option is disabled are not bounds checked. The "malloc" sub-option cannot be controlled at statement level, as it only extracts an alternative malloc implementation from the library.

section all "*name*" (*)
section *type* "*name*"
section_name_with_module

section_name_with_symbol (*)

Changes section names. See Section 1.12, *Compiler Generated Sections* and C compiler option **--rename-sections** for more information.

section code_init | const_init | vector_init

At startup copies corresponding sections to RAM for initialization. See Section 1.12.2, *Influence Section Definition*.

section data_overlay

Allow overlaying data sections.

source / nosource [on | off | default | restore] (*)

With these pragmas you can choose which C source lines must be listed as comments in assembly output.

See C compiler option **--source**.

stdinc [on | off | default | restore] (*)

This pragma changes the behavior of the #include directive. When set, the C compiler options **--include-directory** and **--no-stdinc** are ignored.

switch auto | jump_tab | linear | lookup | default | restore (*)

With these pragmas you can overrule the C compiler chosen switch method.

See Section 1.10, *Switch Statement* and C compiler option **--switch**.

tradeoff *level* [default | restore] (*)

Specify tradeoff between speed (0) and size (4). See C compiler option **--tradeoff**

unroll_factor *value* / endunroll_factor [default | restore] (*)

Specify how many times the following loop should be unrolled, if possible. At the end of the loop use endunroll_factor.

See C compiler option **--unroll-factor**.

user_mode user-0 | user-1 | kernel | default | restore (*)

With this pragma you specify the user mode (I/O privilege mode) the TriCore runs in.

See C compiler option **--user-mode**.

warning [*number*,...] [default | restore] (*)

With this pragma you can disable warning messages. If you do not specify a warning number, all warnings will be suppressed.

weak *symbol*

Mark a symbol as "weak" (.WEAK assembler directive). The symbol must have external linkage, which means a global or external object or function. A static symbol cannot be declared weak.

A weak external reference is resolved by the linker when a global (or weak) definition is found in one of the object files. However, a weak reference will not cause the extraction of a module from a library to resolve the reference. When a weak external reference cannot be resolved, the null pointer is substituted.

A weak definition can be overruled by a normal global definition. The linker will not complain about the duplicate definition, and ignore the weak definition.

1.9. Predefined Preprocessor Macros

The TASKING C compiler supports the predefined macros as defined in the table below. The macros are useful to create conditional C code.

Macro	Description
__BUILD__	Identifies the build number of the compiler, composed of decimal digits for the build number, three digits for the major branch number and three digits for the minor branch number. For example, if you use build 1.22.1 of the compiler, __BUILD__ expands to 1022001. If there is no branch number, the branch digits expand to zero. For example, build 127 results in 127000000.
__CORE_*core*__	A symbol is defined depending on the option **--core=***core*. The *core* is converted to uppercase and '.' is removed. For example, if **--core=tc1.3.1** is specified, the symbol __CORE_TC131__ is defined. When no **--core** is supplied, the compiler defines __CORE_TC13__.
__CTC__	Identifies the compiler. You can use this symbol to flag parts of the source which must be recognized by the TASKING **ctc** compiler only. It expands to 1.
__CPU__	Expands to the name of the CPU supplied with the control program option **--cpu=***cpu*. When no **--cpu** is supplied, this symbol is not defined. For example, if **--cpu=tc1796b** is specified to the control program, the symbol __CPU__ expands to tc1796b.
__CPU_*cpu*__	A symbol is defined depending on the control program option **--cpu=***cpu*. The *cpu* is converted to uppercase. For example, if **--cpu=tc1796b** is specified to the control program, the symbol __CPU_TC1796B__ is defined. When no **--cpu** is supplied, this symbol is not defined.
__DATE__	Expands to the compilation date: "mmm dd yyyy".

Macro	Description
__DOUBLE_FP__	Expands to 1 if you used option **--fp-model=-float**, otherwise unrecognized as macro.
__DSPC__	Indicates conformation to the DSP-C standard. It expands to 1.
__DSPC_VERSION__	Expands to the decimal constant 200001L.
__FILE__	Expands to the current source file name.
__FPU__	Expands to 1 when the option **--fp-model=+soft** is not used. Otherwise unrecognized as a macro.
__LINE__	Expands to the line number of the line where this macro is called.
__MISRAC_VERSION__	Expands to the MISRA C version used 1998, 2004 or 2012 (option **--misrac-version**). The default is 2004.
__PROF_ENABLE__	Expands to 1 if profiling is enabled, otherwise expands to 0.
__REVISION__	Expands to the revision number of the compiler. Digits are represented as they are; characters (for prototypes, alphas, betas) are represented by -1. Examples: v1.0r1 -> 1, v1.0rb -> -1
__SFRFILE__(*cpu*)	If control program option **--cpu=***cpu* is specified, this macro expands to the filename of the used SFR file, including the pathname and the < >. The *cpu* is the argument of the macro. For example, if **--cpu=tc1796b** is specified, the macro `__SFRFILE__(__CPU__)` expands to `__SFRFILE__(tc1796b)`, which expands to `<sfr/regtc1796b.sfr>`.
__SINGLE_FP__	Expands to 1 if you used option **--fp-model=+float** (Treat 'double' as 'float'), otherwise unrecognized as macro.
__STDC__	Identifies the level of ANSI standard. The macro expands to 1 if you set option **--language** (Control language extensions), otherwise expands to 0.
__STDC_HOSTED__	Always expands to 0, indicating the implementation is not a hosted implementation.
__STDC_VERSION__	Identifies the ISO-C version number. Expands to 199901L for ISO C99 or 199409L for ISO C90.
__TASKING__	Identifies the compiler as a TASKING compiler. Expands to 1 if a TASKING compiler is used.
__TIME__	Expands to the compilation time: "hh:mm:ss"
__VERSION__	Identifies the version number of the compiler. For example, if you use version 2.1r1 of the compiler, __VERSION__ expands to 2001 (dot and revision number are omitted, minor version number in 3 digits).

Example

```
#ifdef __FPU__
/* this part is only valid if an FPU is present */
...
#endif
```

1.10. Switch Statement

The TASKING C compiler supports three ways of code generation for a switch statement: a jump chain (linear switch), a jump table or a lookup table.

A *jump chain* is comparable with an if/else-if/else-if/else construction. A *jump table* table filled with target addresses for each possible switch value. The switch argument is used as an index within this table. A *lookup table* is a table filled with a value to compare the switch argument with and a target address to jump to. A binary search lookup is performed to select the correct target address.

By default, the compiler will automatically choose the most efficient switch implementation based on code and data size and execution speed. With the C compiler option **--tradeoff** you can tell the compiler to put more emphasis on speed than on ROM size.

Especially for large switch statements, the jump table approach executes faster than the lookup table approach. Also the jump table has a predictable behavior in execution speed: independent of the switch argument, every case is reached in the same execution time. However, when the case labels are distributed far apart, the jump table becomes sparse, wasting code memory. The compiler will not use the jump table method when the waste becomes excessive.

With a small number of cases, the jump chain method can be faster in execution and shorter in size.

Note that a jump table or lookup table is part of a function and as such is considered code instead of data.

How to overrule the default switch method

You can overrule the compiler chosen switch method by using a pragma:

`#pragma switch linear`   force jump chain code

`#pragma switch jumptab`   force jump table code

`#pragma switch lookup`   force lookup table code

`#pragma switch auto`   let the compiler decide the switch method used (this is the default)

`#pragma switch restore`   restore previous switch method

The switch pragmas must be placed before the `switch` statement. Nested `switch` statements use the same switch method, unless the nested `switch` is implemented in a separate function which is preceded by a different switch pragma.

Example:

```
/* place pragma before function body */

#pragma switch jumptab

void test(unsigned char val)
{ /* function containing the switch */
    switch (val)
    {
        /* use jump table */
```

```
    }
}
```

On the command line you can use C compiler option **--switch**.

1.11. Functions

1.11.1. Calling Convention

Parameter passing

A lot of execution time of an application is spent transferring parameters between functions. The fastest parameter transport is via registers. Therefore, function parameters are first passed via registers. If no more registers are available for a parameter, the compiler pushes parameters on the stack.

Registers available for parameter passing are D4, D5, E4, D6, D7, E6, A4, A5, A6, A7. Up to 4 arithmetic types and 4 pointers can be passed this way. A 64-bit argument is passed in an even/odd data register pair. Parameter registers skipped because of alignment for a 64-bit argument are used by subsequent 32-bit arguments. Any remaining function arguments are passed on the stack. Stack arguments are pushed in reversed order, so that the first one is at the lowest address. On function entry, the first stack parameter is at the address (SP+0).

Structures and unions up to eight bytes are passed via a data register or data register pair. Larger structures/unions are passed via the stack.

All function arguments passed on the stack are aligned on a multiple of 4 bytes. As a result, the stack offsets for all types except float are compatible with the stack offsets used by a function declared without a prototype.

Examples:

```
void func1( int i, char * p, char c );      /* D4 A4 D5 */
void func2( int i1, double d, int i2 );     /* D4 E6 D5 */
void func3( char c1, char c2, char c3[] );  /* D4 D5 A4 */
void func4( double d1, int i1, double d2, int i2 );
                                            /* E4 D6 stack D7 */
```

Function return values

The C compiler uses registers to store C function return values, depending on the function return types.

Return Type	Register
Arithmetic, structure or union <= 32 bits	D2
Arithmetic, structure or union <= 64 bits	D2/D3 (E2)
Pointer	A2
Circular pointer	A2/A3

When the function returns an arithmetic, structure or union type larger than 64 bits, it is copied to a "return area" that is allocated by the caller. The address of this area is passed as an implicit first argument in A4.

Stack usage

The user stack on TriCore derivatives is used for parameter passing and the allocation of automatic and temporary storage. The stack grows from higher addresses to lower addresses. The stack pointer (SP) points to the bottom (low address) of the stack frame. The stack pointer alignment is 8 bytes. For more information about the stack and frame layout refer to section *2.2.2 Stack Frame Management* in the EABI.

Stack model: __stackparm

The function qualifier __stackparm changes the standard calling convention of a function into a convention where all function arguments are passed via the stack, conforming a so-called stack model. This qualifier is only needed for situations where you need to use an indirect call to a function for which you do not have a valid prototype.

> Note that the TASKING TriCore compiler deviates from the EABI at this point. The EABI states that objects larger than 64 bits must be passed via a pointer and a copy of the object is not necessary. This is dangerous, because the user is then responsible for the copy object (if required). Therefore, the TASKING TriCore compiler places ALL arguments on the stack.

The compiler sets the least significant bit of the function pointer when you take the address of a function declared with the __stackparm qualifier, so that these function pointers can be identified at run-time. The least significant bit of a function pointer address is ignored by the hardware.

Example:

```
void             plain_func ( int );
void __stackparm stack_func ( int );

void call_indirect ( unsigned int fp, int arg )
{
    typedef __stackparm void (*SFP)( int );
    typedef void (*RFP)( int );

    SFP     fp_stack;
    RFP     fp_reg;

    if ( fp & 1 )
    {
        fp_stack = (SFP) fp;
        fp_stack( arg );
    }
    else
    {
        fp_reg = (RFP) fp;
        fp_reg( arg );
    }
```

```
}

void main ( void )
{
     call_indirect( (unsigned int) plain_func, 1 );
     call_indirect( (unsigned int) stack_func, 2 );
}
```

Function calling modes: __indirect

Functions are by default called with a single word direct call. However, when you link the application and the target address appears to be out of reach (+/- 16 MB from the `callg` or `jg` instruction), the linker generates an error. In this case you can use the `__indirect` keyword to force the less efficient, two and a half word indirect call to the function:

```
int __indirect foo( void )
{
...
}
```

With C compiler option **--indirect** you tell the C compiler to generate far calls for *all* functions.

1.11.2. Register Usage

The C compiler uses the data registers and address registers according to the convention given in the following table.

Register		Usage	Register	Usage
D0	E0	scratch	A0	global
D1		scratch	A1	global
D2	E2	return register for arithmetic types and struct/union	A2	return register for pointers
D3		most significant part of 64 bit result	A3	scratch
D4	E4	parameter passing	A4	parameter passing
D5		parameter passing	A5	parameter passing
D6	E6	parameter passing	A6	parameter passing
D7		parameter passing	A7	parameter passing
D8	E8	saved register	A8	global
D9		saved register	A9	global
D10	E10	saved register	A10	stack pointer
D11		saved register	A11	link register
D12	E12	saved register	A12	saved register
D13		saved register	A13	saved register
D14	E14	saved register	A14	saved register

Register		Usage	Register	Usage
D15		saved register, implicit pointer	A15	saved register, implicit pointer

1.11.3. Inlining Functions: inline

With the C compiler option **--optimize=+inline**, the C compiler automatically inlines small functions in order to reduce execution time (smart inlining). The compiler inserts the function body at the place the function is called. The C compiler decides which functions will be inlined. You can overrule this behavior with the two keywords `inline` (ISO-C) and `__noinline`.

With the `inline` keyword you force the compiler to inline the specified function, regardless of the optimization strategy of the compiler itself:

```
inline unsigned int abs(int val)
{
    unsigned int abs_val = val;
    if (val < 0) abs_val = -val;
    return abs_val;
}
```

If a function with the keyword `inline` is not called at all, the compiler does not generate code for it.

You must define inline functions in the same source module as in which you call the function, because the compiler only inlines a function in the module that contains the function definition. When you need to call the inline function from several source modules, you must include the definition of the inline function in each module (for example using a header file).

With the `__noinline` keyword, you prevent a function from being inlined:

```
__noinline unsigned int abs(int val)
{
    unsigned int abs_val = val;
    if (val < 0) abs_val = -val;
    return abs_val;
}
```

Using pragmas: inline, noinline, smartinline

Instead of the `inline` qualifier, you can also use `#pragma inline` and `#pragma noinline` to inline a function body:

```
#pragma inline
unsigned int abs(int val)
{
    unsigned int abs_val = val;
    if (val < 0) abs_val = -val;
    return abs_val;
}
#pragma noinline
void main( void )
```

```
{
    int i;
    i = abs(-1);
}
```

If a function has an `inline`/`__noinline` function qualifier, then this qualifier will overrule the current pragma setting.

With the `#pragma noinline`/`#pragma smartinline` you can temporarily disable the default behavior that the C compiler automatically inlines small functions when you turn on the C compiler option **--optimize=+inline**.

With the C compiler options **--inline-max-incr** and **--inline-max-size** you have more control over the automatic function inlining process of the compiler.

Combining inline with __asm to create intrinsic functions

With the keyword `__asm` it is possible to use assembly instructions in the body of an inline function. Because the compiler inserts the (assembly) body at the place the function is called, you can create your own intrinsic function. See Section 1.11.5, *Intrinsic Functions*.

1.11.4. Interrupt and Trap Functions

The TriCore C compiler supports a number of function qualifiers and keywords to program interrupt service routines (ISR) or trap handlers. Trap handlers may also be defined by the operating system if your target system uses one.

An *interrupt service routine* (or: interrupt function, or: interrupt handler) is called when an interrupt event (or: *service request*) occurs. This is always an external event; peripherals or external inputs can generate an interrupt signals to the CPU to request for service. Unlike other interrupt systems, each interrupt has a unique *interrupt request priority number* (IRPN). This number (0 to 255) is set as the pending interrupt priority number (PIPN) in the interrupt control register (ICR) by the interrupt control unit. If multiple interrupts occur at the same time, the priority number of the request with the highest priority is set, so this interrupt is handled.

The TriCore vector table provides an entry for each pending interrupt priority number, not for a specific interrupt source. A request is handled if the priority number is higher than the CPU priority number (CCPN). An interrupt service routine can be interrupted again by another interrupt request with a higher priority. Interrupts with priority number 0 are never handled.

A trap service routine (or: trap function, or: trap handler) is called when a trap event occurs. This is always an event generated within or by the application. For example, a divide by zero or an invalid memory access.

For information on locating the interrupt vector table and trap vector table and the possibility to copy the vector tables from ROM to RAM at startup, see subsection Vector table in Section 16.4.3, *Defining Address Spaces* in Chapter 16, *Linker Script Language (LSL)*.

Overview of function qualifiers

With the following function qualifiers you can declare an interrupt handler or trap handler:

```
__interrupt()    __interrupt_fast()
__interrupt8()   __interrupt8_fast()
__trap()         __trap_fast()
__vector_table()
```

There is one special type of trap function which you can call manually, the system call exception (trap class 6). See Section 1.11.4.3, *Defining a Trap Service Routine Class 6: __syscallfunc()*.

```
__syscallfunc()
```

During the execution of an interrupt service routine or trap service routine, the system blocks the CPU from taking further interrupt requests. With the following keywords you can enable interrupts again, immediately after an interrupt or trap function is called:

```
__enable_        __bisr_()
```

Multi-core interrupt/trap vector table number

For TriCore 1.6.x or 1.6.2 derivatives that support multiple cores, an interrupt vector table and trap vector table is present for each core. These vector tables are defined in the LSL files `inttab`*nr*`.lsl` and `traptab`*nr*`.lsl`. The vector table number *nr* corresponds to the TriCore core used. The default core is `tc0` and therefore, the default vector table number is 0.

The compiler generates a vector table entry for an interrupt function or trap function. With the interrupt function qualifier `__vector_table` you can assign it to one or more core vector table.

Syntax:

```
__vector_table(vector_table_number,...)
```

When you do not specify `__vector_table` for an interrupt function or trap function, the default vector table number 0 is used for functions with a clone or share code association. See also Section 1.4.2, *Code Core Association*.

You do not have to specify a vector table number for an interrupt function or trap function with a private code core association, the vector table corresponds to the private core number association. When you do specify a vector table number for an interrupt function or trap function with a private code core association, the number must correspond to the private core number association.

Fast interrupt functions or fast trap functions are only allowed for functions that have a share code core association and can only be assigned to one vector table.

Restrictions of `__vector_table`:

• `__vector_table` is only allowed for multi-core TriCore derivatives.

• `__vector_table` is only allowed for (fast) interrupt functions or trap qualified functions.

• `__vector_table` does not accept more vector table numbers than defined by the number of cores in the TriCore architecture.

• `__vector_table` does not accept duplicate vector table numbers.

1.11.4.1. Defining an Interrupt Service Routine: __interrupt(), __interrupt_fast(), __interrupt8(), __interrupt8_fast()

With the function type qualifier `__interrupt()` you can declare a function as an interrupt service routine. The function type qualifier `__interrupt()` takes one interrupt vector (0..255) as argument.

Interrupt functions cannot return anything and must have a void argument type list:

```
void __interrupt(vector) [__vector_table(nr,...)]
isr( void )
{
...
}
```

The argument *vector* identifies the entry into the interrupt vector table (0..255). Unlike other interrupt systems, the priority number (PIPN) of the interrupt now being serviced by the CPU identifies the entry into the vector table.

For an extensive description of the TriCore interrupt system, see the *TriCore 1 Unified Processor Core v1.3 Architecture Manual, Doc v1.3.3* [2002-09, Infineon].

The compiler generates an interrupt service frame for interrupts. The difference between a normal function and an interrupt function is that an interrupt function ends with an RFE instruction instead of a RET, and that the lower context is saved and restored with a pair of SVLCX / RSLCX instructions when one of the lower context registers is used in the interrupt handler.

When you define an interrupt service routine with the `__interrupt()` qualifier, the compiler generates an entry for the interrupt vector table. This vector jumps to the interrupt handler.

The compiler puts the interrupt vectors in sections with the following naming convention:

```
.text[.inttab{0|1|2}].intvec.vector
```

The optional `.inttab0`, `.inttab1` or `.inttab2` is generated when one of the cores of the TriCore 1.6.x or 1.6.2 is selected. You can specify a core vector table by using the interrupt function qualifier `__vector_table`.

The following example illustrates the function definition for a function for a software interrupt with vector number 0x30:

```
int c;

void __interrupt( 0x30 ) __vector_table( 1 ) transmit( void )
{
    c = 1;
}
```

This results in a section called "`.text.inttab1.intvec.030`".

8 byte vector table entry support (TriCore 1.6.x and 1.6.2 only)

For the TriCore 1.6.x and 1.6.2 an entry in the vector table can be 32 bytes or 8 bytes. For 32 byte entries you can use `__interrupt()` as explained above. For 8 byte vector table entries you can use function type qualifier `__interrupt8()`.

```
void __interrupt8(vector) [__vector_table(nr,...)]
isr8( void )
{
...
}
```

An absolute jump instruction is generated to the interrupt service routine, which restricts the address range to absolute 24. Loading a 32-bit address and jumping indirectly does not fit in an 8 byte vector.

The compiler puts the 8 byte interrupt vectors in sections with the following naming convention:

```
.text[.inttab{0|1|2}].intvec8.vector
```

8 byte and 32 byte spacing is available at the same time, no LSL configuration is required. Mixing 8 byte and 32 byte spacing on the same core is not possible, but different cores can use different spacings. You define at compile which kind of spacing is required.

The vector spacing is configured at startup per core in the Base Interrupt Vector (BIV) with startup macro `__BIV_8BYTE_INIT` (see `cstart.c`). It is your responsibility that this is conform the spacing required by the interrupt functions, because the compiler cannot check if usage of interrupt functions qualifiers corresponds with the BIV configuration. In Eclipse you can set this macro as follows:

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Startup Configuration**.

    In the right pane the Startup Configuration page appears.

3. Enable the option **Initialize 8 byte spacing interrupt vector table**.

4. Click **OK**.

    The file cstart.h in your project is updated with the new value.

Fast interrupts

When you define an interrupt service routine with the `__interrupt_fast()` qualifier, the interrupt handler is directly placed in the interrupt vector table, thereby eliminating the jump code. There is only 32 bytes of space available for an entry in the vector table, but the compiler does not check this restriction. Fast interrupts can span more than one vector. Fast interrupts are only restricted to one entry when the next interrupt vector is also occupied. The linker generates an error when the fast interrupt does not fit or overlaps with another vector or interrupt.

For the TriCore 1.6.x and 1.6.2 an entry in the vector table can be 32 bytes or 8 bytes using Base Interrupt Vector 0 (BIV[0]). For 8 byte fast interrupts you use the `__interrupt8_fast()` qualifier.

1.11.4.2. Defining a Trap Service Routine: __trap(), __trap_fast()

The definition of a trap service routine is similar to the definition of an interrupt service routine. Trap functions cannot accept arguments and do not return anything:

```
void __trap( class ) [__vector_table(nr,...)]
tsr( void )
{
...
}
```

The argument *class* identifies the entry into the trap vector table. TriCore defines eight classes of trap functions. Each class has its own trap handler.

When a trap service routine is called, the d15 register contains the so-called *Trap Identification Number* (TIN). This number identifies the cause of the trap. In the trap service routine you can test and branch on the value in d15 to reach the sub-handler for a specific TIN. With the intrinsic function `__get_tin()` you can use the TIN anywhere in your code.

The following table shows the classes supported by TriCore.

Class	Description
Class 0	Reset
Class 1	Internal Protection Traps
Class 2	Instruction Errors
Class 3	Context Management
Class 4	System Bus and Peripheral Errors
Class 5	Assertion Traps
Class 6	System Call
Class 7	Non-Maskable Interrupt

For a complete overview of the trap system and the meaning of the trap identification numbers, see the *TriCore 1 Unified Processor Core v1.3 Architecture Manual, Doc v1.3.3* [2002-09, Infineon]

Analogous to interrupt service routines, the compiler generates a trap service frame for interrupts.

When you define a trap service routine with the `__trap()` qualifier, the compiler generates an entry for the interrupt vector table. This vector jumps to the trap handler.

The compiler puts the trap vectors in sections with the following naming convention:

`.text[.traptab{0|1|2}].trapvec.class`

The optional `.traptab0`, `.traptab1` or `.traptab2` is generated when one of the cores of the TriCore 1.6.x or 1.6.2 is associated with the trap vector. You can specify a core vector table by using the interrupt function qualifier `__vector_table`.

The following example illustrates the function definition for a reset trap:

```
int c;

void __trap( 0 ) __vector_table( 1 ) rst( void )
{
    c = 1;
}
```

This results in a section called ".`text.traptabl.trapvec.000`".

Fast traps

When you define a trap service routine with the `__trap_fast()` qualifier, the trap handler is directly placed in the trap vector table, thereby eliminating the jump code. You should only use this when the trap handler is very small, as there is only 32 bytes of space available in the vector table. The compiler does not check this restriction.

1.11.4.3. Defining a Trap Service Routine Class 6: __syscallfunc()

A special kind of trap service routine is the system call trap. With a system call the trap service routine of class 6 is called. For the system call trap, the trap identification number (TIN) is taken from the immediate constant specified with the function qualifier `__syscallfunc()`:

```
__syscallfunc(TIN)
```

The TIN is a value in the range 0 and 255. You can only use `__syscallfunc()` in the function declaration. A function body is useless, because when you call the function declared with `__syscallfunc()`, a trap class 6 occurs which calls the corresponding trap service routine.

> In case of the other traps, when a trap service routine is called, the system places a trap identification number in `d15`.

Unlike the other traps, a class 6 trap service routine can contain arguments and return a value (the lower context is not saved and restored). Arguments that are passed via the stack, remain on the stack of the caller because it is not possible to pass arguments from the user stack to the interrupt stack on a system call. This restriction, caused by the TriCore's run-time behavior, cannot be checked by the compiler.

Example

The following example illustrates the definition of a class 6 trap service routine and the corresponding system call:

```
__syscallfunc(1) int syscall_a( int, int );
__syscallfunc(2) int syscall_b( int, int );

int x;

void main( void )
{
    x = syscall_a(1,2);    // causes a trap class 6 with TIN = 1
    x = syscall_b(4,3);    // causes a trap class 6 with TIN = 2
```

```
}

int __trap( 6 ) trap6( int a, int b ) // trap class 6 handler
{
    int tin;
    tin = __get_tin(); // get the TIN

    switch( tin )
    {
    case 1:
        a += b;
        break;
    case 2:
        a -= b;
        break;
    default:
        break;
    }
    return a;
}
```

1.11.4.4. Enabling Interrupt Requests: __enable_, __bisr_()

Enabling interrupt service requests

During the execution of an interrupt service routine or trap service routine, the system blocks the CPU from taking further interrupt requests. You can immediately re-enable the system to accept interrupt requests:

```
__interrupt(vector) __enable_ isr( void )
__trap(class) __enable_ tsr( void )
```

The compiler generates an `enable` instruction as first instruction in the routine. The `enable` instruction sets the interrupt enable bit (ICR.IE) in the interrupt control register.

You can also generate the `enable` instruction with the intrinsic function __enable(), but it is not guaranteed that it will be the first instruction in the routine.

Enabling interrupt service requests and setting CPU priority number

The function qualifier __bisr_() also re-enables the system to accept interrupt requests. In addition, the *current CPU priority number* (CCPN) in the interrupt control register is set:

```
__interrupt(vector) __bisr_(CCPN) isr( void )
__trap(class) __bisr_(CCPN) tsr( void )
```

The argument *CCPN* is a number between 0 and 255. The system accepts all interrupt requests that have a higher pending interrupt priority number (PIPN) than the current CPU priority number. So, if the CPU priority number is set to 0, the system accepts all interrupts. If it is set to 255, no interrupts are accepted.

The compiler generates a `bisr` instruction as first instruction in the routine. The `bisr` instruction sets the interrupt enable bit (ICR.IE) and the current CPU priority number (ICR.CCPN) in the interrupt control register.

You can also generate the `bisr` instruction with the intrinsic function `__bisr()`, but it is not guaranteed that it will be the first instruction in the routine.

Setting the CPU priority number in a Class 6 trap service routine

The `bisr` instruction saves the lower context so passing and returning arguments is not possible. Therefore, you cannot use the function qualifier `__bisr_()` for class 6 traps.

Instead, you can use the function qualifier `__enable_` to set the ICR.IE bit, and the intrinsic function `__mtcr( int, int )` to set the ICR.CCPN value at the beginning of a class 6 trap service routine (or use the intrinsic function `__mtcr()` to set both the ICR.IE bit and the ICR.CCPN value).

1.11.4.5. Single Entry Vector Table for TriCore 1.6.x and 1.6.2

For the TriCore 1.6.x and 1.6.2 you can reduce the vector table to a single entry by masking the PIPN.

A minimum vector table can be configured if the BIV masks the PIPN so that any interrupt address calculation results in the same address.

For example in `cstart.c`:

```
__mtcr(BIV, (unsigned int)(_lc_u_int_tab) | (0xff<<3) | 1 );
```

This configures the BIV register to use a common, single entry where a function interrupt handler is located to branch to the specific interrupt routine by using an array of function pointers. A pointer to an array is used to switch the array quickly.

The C library contains functions to support Single Entry Vector Table (SEVT) for TriCore1.6.x and 1.6.2. Interrupt Service Routines can be installed in the SEVT ISR array for each core, using `_sevt_isr_install`. For example, install C function `blink()` with Interrupt Request Priority Number (IRPN) 1 on core tc0.

```
#include <sevt.h>

extern void blink( void );
_sevt_isr_install( 1, &blink, 0 );
```

The SEVT ISR handler indirectly calls the functions installed in the SEVT data array. The SEVT ISR handler is located at interrupt vector table entry 64. The SEVT ISR handler and SEVT data array are supported by `_sevt_isr_tc0|1|2()` and `_sevt_isrs_tc0|1|2[]` in C library module `sevt.c`. The SEVT data array can be switched with `_sevt_isr_install_array()`. SEVT can be enabled by cstart macro `__BIV_SINGLE_INIT`, `__BIV_SINGLE_TC1_INIT` and `__BIV_SINGLE_TC2_INIT` (see files `cstart*.h`).

1.11.5. Intrinsic Functions

Some specific assembly instructions have no equivalence in C. *Intrinsic functions* give the possibility to use these instructions. Intrinsic functions are predefined functions that are recognized by the compiler. The compiler generates the most efficient assembly code for these functions.

The compiler always inlines the corresponding assembly instructions in the assembly source (rather than calling it as a function). This avoids parameter passing and register saving instructions which are normally necessary during function calls.

Intrinsic functions produce very efficient assembly code. Though it is possible to inline assembly code by hand, intrinsic functions use registers even more efficiently. At the same time your C source remains very readable.

You can use intrinsic functions in C as if they were ordinary C (library) functions. All intrinsics begin with a double underscore character (__).

The following example illustrates the use of an intrinsic function and its resulting assembly code.

```
x = __min( 4,5 );
```

The resulting assembly code is inlined rather than being called:

```
mov16  d2,#4
min    d2,d2,#5
```

The intrinsics cover the following subjects:

* Minimum and maximum of (short) integers

* Fractional data type support

* Packed data type support

* Interrupt handling

* Insert single assembly instruction

* Register handling

* Insert / extract bit-fields and bits

* Miscellaneous

Writing your own intrinsic function

Because you can use any assembly instruction with the `__asm()` keyword, you can use the `__asm()` keyword to create your own intrinsic functions. The essence of an intrinsic function is that it is inlined.

1. First write a function with assembly in the body using the keyword `__asm()`. See Section 1.6, *Using Assembly in the C Source: __asm()*

2. Next make sure that the function is inlined rather than being called. You can do this with the function qualifier `inline`. This qualifier is discussed in more detail in Section 1.11.3, *Inlining Functions: inline*.

```
int a, b, result;

inline void __my_mul( void )
{
    __asm( "mul %0, %1, %2": "=d"(result): "d"(a), "d"(b) );
}

void main(void)
{
    // call to function __my_mul
    __my_mul();
}
```

Generated assembly code:

```
main:
    ; __my_mul code is inlined here
    ld.w  d15,a
    ld.w  d0,b
    mul   d15, d15, d0
    st.w  result,d15
```

As you can see, the generated assembly code for the function `__my_mul` is inlined rather than called.

1.11.5.1. Minimum and Maximum of (Short) Integers

The following table provides an overview of the intrinsic functions that return the minimum or maximum of a signed integer, unsigned integer or short integer.

Intrinsic Function	Description
int __min(int, int);	Return minimum of two integers
short __mins(short, short);	Return minimum of two short integers
unsigned int __minu(unsigned int, unsigned int);	Return minimum of two unsigned integers
int __max(int, int);	Return maximum of two integers
short __maxs(short, short);	Return maximum of two short integers
unsigned int __maxu(unsigned int, unsigned int);	Return maximum of two unsigned integers

1.11.5.2. Fractional Arithmetic Support

The following table provides an overview of intrinsic functions to convert fractional values. Note that the TASKING VX-toolset C compiler for TriCore fully supports the fractional type so normally you should not need these intrinsic functions (except for `__mulfractlong`). For compatibility reasons the TASKING C compiler does support these functions.

Conversion of fractional values

Intrinsic Function	Description
long __mulfractlong(__fract, long);	Integer part of the multiplication of a `__fract` and a `long`
__sfract __round16(__fract);	Convert `__fract` to `__sfract`
__fract __getfract(__laccum);	Convert `__laccum` to `__fract`
short __clssf(__sfract);	Count the consecutive number of bits that have the same value as bit 15 of an `__sfract`
__sfract __shasfracts(__sfract, int);	Left/right shift of an `__sfract`
__fract __shafracts(__fract, int);	Left/right shift of an `__fract`
__laccum __shaaccum(__laccum, int);	Left/right shift of an `__laccum`
__sfract __mac_sf(__sfract a, __sfract b, __sfract c);	Multiply-add `__sfract`. Returns (a + b * c)
__sfract __mac_r_sf(__sfract, __sfract, __sfract);	Multiply-add with rounding. Returns the rounded result of (a + b * c)
__sfract __u16_to_sfract(unsigned short integer);	Convert `unsigned short` to `__sfract`
__sfract __s16_to_sfract(signed short integer);	Convert `signed short` to `__sfract`
unsigned short int __sfract_to_u16(__sfract);	Convert `__sfract` to `unsigned short`
signed short int __sfract_to_s16(__sfract);	Convert `__sfract` to `signed short`

1.11.5.3. Packed Data Type Support

The following table provides an overview of the intrinsic functions for initialization of packed data type.

Initialize packed data types

Intrinsic Function	Description
__packb __initpackbl(long);	Initialize `__packb` with a long integer
__packb __initpackb(int, int, int, int);	Initialize `__packb` with four integers
unsigned __packb __initupackb(unsigned, unsigned, unsigned, unsigned);	Idem, but unsigned
__packhw __initpackhwl(long);	Initialize `__packhw` with a long integer
__packhw __initpackhw(short, short);	Initialize `__packhw` with two integers
unsigned __packhw __initupackhw(unsigned short, unsigned short);	Idem, but unsigned

Extract values from packed data types

The following table provides an overview of the intrinsic functions to extract a single byte or halfword from a `__packb` or `__packhw` data type.

Intrinsic Function	Description
char __extractbyte1(__packb);	Extract first byte from a `__packb`
unsigned char __extractubyte1(__unsigned packb);	Idem, but unsigned
char __extractbyte2(__packb);	Extract second byte from a `__packb`
unsigned char __extractubyte2(__unsigned packb);	Idem, but unsigned
char __extractbyte3(__packb);	Extract third byte from a `__packb`
unsigned char __extractubyte3(__unsigned packb);	Idem, but unsigned
char __extractbyte4(__packb);	Extract fourth byte from a `__packb`
unsigned char __extractubyte4(__unsigned packb);	Idem, but unsigned
short __extracthw1(__packhw);	Extract first short from a `__packhw`
unsigned short __extractuhw1(unsigned __packhw);	Idem, but unsigned
short __extracthw2(__packhw);	Extract second short from a `__packhw`
unsigned short __extractuhw2(unsigned __packhw);	Idem, but unsigned
volatile char __getbyte1(__packb *);	Extract first byte from a `__packb`
volatile unsigned char __getubyte1(__unsigned packb *);	Idem, but unsigned
volatile char __getbyte2(__packb *);	Extract second byte from a `__packb`
volatile unsigned char __getubyte2(__unsigned packb *);	Idem, but unsigned
volatile char __getbyte3(__packb *);	Extract third byte from a `__packb`
volatile unsigned char __getubyte3(__unsigned packb *);	Idem, but unsigned
volatile char __getbyte4(__packb *);	Extract fourth byte from a `__packb`
volatile unsigned char __getubyte4(__unsigned packb *);	Idem, but unsigned
volatile short __gethw1(__packhw *);	Extract first short from a `__packhw`
volatile unsigned short __getuhw1(unsigned __packhw *);	Idem, but unsigned
volatile short __gethw2(__packhw *);	Extract second short from a `__packhw`
volatile unsigned short __getuhw2(unsigned __packhw *);	Idem, but unsigned

Insert values into packed data types

The following table provides an overview of the intrinsic functions to insert a single byte or halfword into a `__packb` or `__packhw` data type.

Intrinsic Function	Description
__packb __insertbyte1(__packb, char);	Insert char into first byte of a `__packb`
unsigned __packb __insertubyte1(unsigned __packb, unsigned char);	Idem, but unsigned
__packb __insertbyte2(__packb, char);	Insert char into second byte of a `__packb`
unsigned __packb __insertubyte2(unsigned __packb, unsigned char);	Idem, but unsigned

Intrinsic Function	Description
__packb __insertbyte3(__packb, char);	Insert char into third byte of a `__packb`
unsigned __packb __insertubyte3(unsigned __packb, unsigned char);	Idem, but unsigned
__packb __insertbyte4(__packb, char);	Insert char into fourth byte of a `__packb`
unsigned __packb __insertubyte4(unsigned __packb, unsigned char);	Idem, but unsigned
__packhw __inserthw1(__packhw, short);	Insert short into first halfword of a `__packhw`
unsigned __packhw __insertuhw1(unsigned __packhw, unsigned short);	Idem, but unsigned
__packhw __inserthw2(__packhw, short);	Insert short into second halfword of a `__packhw`
unsigned __packhw __insertuhw2(unsigned __packhw, unsigned short);	Idem, but unsigned
void __setbyte1(__packb *, char);	Insert char into first byte of a `__packb`
void __setubyte1(unsigned __packb *, unsigned char);	Idem, but unsigned
void __setbyte2(__packb *, char);	Insert char into second byte of a `__packb`
void __setubyte2(unsigned __packb *, unsigned char);	Idem, but unsigned
void __setbyte3(__packb *, char);	Insert char into third byte of a `__packb`
void __setubyte3(unsigned __packb *, unsigned char);	Idem, but unsigned
void __setbyte4(__packb *, char);	Insert char into fourth byte of a `__packb`
void __setubyte4(unsigned __packb *, unsigned char);	Idem, but unsigned
void __sethw1(__packhw *, short);	Insert short into first halfword of a `__packhw`
void __setuhw1(unsigned __packhw *, unsigned short);	Idem, but unsigned
void __sethw2(__packhw *, short);	Insert short into second halfword of a `__packhw`
void __setuhw2(unsigned __packhw *, unsigned short);	Idem, but unsigned

Calculate absolute values of packed data type values

The following table provides an overview of the intrinsic functions to calculate the absolute value of packed data type values.

Intrinsic Function	Description
__packb __absb(__packb);	Absolute value of `__packb`
__packhw __absh(__packhw);	Absolute value of `__packhw`
__sat __packhw __abssh(__sat __packhw);	Absolute value of `__packhw` using saturation

Calculate minimum packed data type values

The following table provides an overview of the intrinsic functions to calculate the minimum from two packed data type values.

Intrinsic Function	Description
__packb __minb(__packb,__packb);	Minimum of two `__packb` values
unsigned __packb __minbu(unsigned __packb, unsigned __packb);	Minimum of two unsigned `__packb` values
__packhw __minh(__packhw,__packhw);	Minimum of two `__packhw` values
unsigned __packhw __minhu(unsigned __packhw, unsigned __packhw);	Minimum of two unsigned `__packhw` values

1.11.5.4. Interrupt Handling

The following table provides an overview of the intrinsic functions to read or set interrupt handling.

Intrinsic Function	Description
volatile void __enable (void);	Enable interrupts immediately at function entry
volatile void __disable (void);	Disable interrupts. Only supported for TriCore1.
volatile int __disable_and_save (void);	Disable interrupts and return previous interrupt state (enabled or disabled). Only supported for TriCore1.6 (**--core=tc1.6**).
unsigned int __get_tin(void);	Get the Trap Identification Number (TIN). See Section 1.11.4.2, *Defining a Trap Service Routine: __trap(), __trap_fast()*.
volatile void __restore (int);	Restore interrupt state. Only supported for TriCore1.6 (**--core=tc1.6**).
volatile void __bisr (const unsigned int);	Set CPU priority number [0..255] and enable interrupts immediately at function entry
volatile void __syscall (int);	Call a system call function number

1.11.5.5. Insert Single Assembly Instruction

The following table provides an overview of the intrinsic functions that you can use to insert a single assembly instruction. You can also use inline assembly but these intrinsics provide a shorthand for frequently used assembly instructions.

See Section 1.6, *Using Assembly in the C Source: __asm()*.

Intrinsic Function	Description
volatile void __debug(void);	Insert DEBUG instruction
volatile void __dsync(void);	Insert DSYNC instruction
volatile void __isync(void);	Insert ISYNC instruction
volatile void __svlcx(void);	Insert SVLCX instruction
volatile void __rslcx(void);	Insert RSLCX instruction

Intrinsic Function	Description
volatile void __nop(void);	Insert NOP instruction
volatile void __ldmst(unsigned int * *address*, unsigned int *mask*, unsigned int *value*);	Insert LDMST instruction. Note that *address* must be word-aligned.
volatile unsigned int __swap(unsigned int * *place*, unsigned int *value*);	Insert SWAP instruction. Note that *place* must be word-aligned.

1.11.5.6. Register Handling

Access control registers

The following table provides an overview of the intrinsic functions that you can use to access control registers.

Intrinsic Function	Description
volatile int __mfcr(int *csfr*);	Move contents of the addressed Core Special Function Register (CSFR) into a data register
volatile void __mtcr (int *csfr*, int *val*);	Move contents of a data register (second int) to the addressed CSFR (first int) and generate an ISYNC instruction. The ISYNC instruction ensures that the effects of the CSFR update are correctly seen by all following instructions.

See the `.sfr` files in the `include\sfr` directory for a list of the 16-bit CSFRs.

For example:

```
#include "sfr/regtc1796b.sfr"

int get_cpu_id( void )
{
    return __mfcr( CPU_ID ); // return contents of CSFR CPU_ID
}
```

This results in the assembly instruction:

```
    mfcr d2,#65048
```

Note that if you want to set a single bit in a CSFR you have to create a bit mask. For example in `cstart.c`:

```
__mtcr(BIV, (unsigned int)(_lc_u_int_tab) | (0xff<<3) | 1 );
```

Perform register value operations

The following table provides an overview of the intrinsic functions that operate on a register and return a value in another register.

Intrinsic Function	Description
int __clz (int);	Count leading zeros in int

Intrinsic Function	Description
int __clo (int);	Count leading ones in int
int __cls (int);	Count number of redundant sign bits (all consecutive bits with the same value as bit 31
signed char __satb (int);	Return saturated byte
unsigned char __satbu (int);	Return saturated unsigned byte
short __sath (int);	Return saturated halfword
unsigned short __sathu (int);	Return saturated unsigned halfword
int __abs (int);	Return absolute value
int __abss (int);	Return absolute value with saturation
float __fabsf (float f);	Return absolute floating-point value
double __fabs (double d);	Return absolute double precision floating-point value
int __parity (int);	Return parity

Get or set stack pointer register A10

The following table provides an overview of the intrinsic functions to set or get the stack pointer register A10.

Intrinsic Function	Description
void __set_sp(void *);	Set stack pointer register
void * __get_sp(void);	Get stack pointer register

Example:

```
# define STACK_ALIGN 0xfffffff8
void set( void )
{
    void * sp = (void *)((unsigned int)(_lc_ue_ustack) & STACK_ALIGN);

    __set_sp( sp );   /* load user stack pointer */
}

void * get( void )
{
    return __get_sp();
}
```

1.11.5.7. Insert / Extract Bit-fields and Bits

Insert / extract bit-fields

The following table provides an overview of the intrinsic functions to insert or extract a bit-field.

Intrinsic Function	Description
int __extr (int *value*, int *pos*, int *width*);	Extract a bit-field (bit *pos* to bit *pos*+*width*) from *value*
unsigned int __extru (int *value*, int *pos*, int *width*);	Same as `__extr()` but return bit-field as unsigned integer
int __insert (int *trg*, int *src*, int *pos*, int *width*);	Extract bit-field (*width* bits starting at bit 0) from *src* and insert it in *trg* at *pos*.
int __ins(int *trg*, int *trgbit*, int *src*, int *srcbit*);	Return *trg* but replace *trgbit* by *srcbit* in *src*.
int __insn(int *trg*, int *trgbit*, int *src*, int *srcbit*);	Return *trg* but replace *trgbit* by inverse of *srcbit* in *src*.

Atomic load-modify-store

With the following intrinsic function you can perform atomic Load-Modify-Store of a bit-field from an integer value. This function uses the IMASK and LDMST instruction. The intrinsic writes the number of *bits* of an integer *value* at a certain *address* location in memory with a *bitoffset*. The number of bits must be a constant value. Note that all operands must be word-aligned.

Intrinsic Function	Description
void __imaskldmst(int* *address*, int *value*, int *bitoffset*, int *bits*);	Atomic load-modify-store

Store a single bit

With the intrinsic macro `__putbit()` you can store a single bit atomicly in memory at a specified bit offset. The bit at offset 0 in *value* is stored at an *address* location in memory with a *bitoffset*.

This intrinsic is implemented as a macro definition which uses the `__imaskldmst()` intrinsic:

```
#define __putbit ( value, address, bitoffset ) __imaskldmst
  ( address, value, bitoffset, 1 )
```

Note that all operands must be word-aligned.

Intrinsic Function	Description
void __putbit(int *value*, int* *address*, int *bitoffset*);	Store a single bit

Load a single bit

With the intrinsic macro `__getbit()` you can load a single bit from memory at a specified bit offset. A bit value is loaded from an *address* location in memory with a *bitoffset* and returned as an unsigned integer value.

This intrinsic is implemented as a macro definition which uses the `__extru()` intrinsic:

```
#define __getbit( address, bitoffset ) __extru( *(address), bitoffset, 1 )
```

Intrinsic Function	Description
unsigned int __getbit(int * *address*, int *bitoffset*);	Load a single bit

1.11.5.8. Miscellaneous Intrinsic Functions

Multiply and scale back

Intrinsic Function	Description
int __mulsc(int a, int b, int *offset*);	Multiply two 32-bit numbers to an intermediate 64-bit result, and scale back the result to 32 bits. To scale back the result, 32 bits are extracted from the intermediate 64-bit result: bit 63-*offset* to bit 31-*offset*.

Cache write back and invalidation

To support write back and invalidation of cache address or cache index the following intrinsics are available:

Intrinsic Function	Description
volatile void __cacheawi(unsigned char * p);	Write back and invalidate cache address "p". Generates CACHEA.WI [Ab].
volatile void __cacheiwi(unsigned char * p);	Write back and invalidate cache index "p". Generates CACHEI.WI [Ab].
unsigned char * __cacheawi_bo_post_inc(unsigned char * p);	Write back and invalidate cache address "p" and return post incremented value of "p". Generates CACHEA.WI [Ab+].

See `sync_on_halt.c` for some examples.

Swap

Intrinsic Function	Description
volatile unsigned int __swapmskw(unsigned int * *memory*, unsigned int *value*, unsigned int *mask*);	Swap under mask. Exchanges the values of *value* and *memory*, but only those bits that are allowed by *mask*. Before the `swapmsk.w` instruction is generated, the parameters *value* and *mask* are moved into a double register. Only supported for TriCore1.6.x (**--core=tc1.6.x**) and TriCore1.6.2 (**--core=tc1.6.2**). Note that *memory* must be word-aligned.
volatile unsigned int __cmpswapw(unsigned int * *memory*, unsigned int *value*, unsigned int *compare*);	Compare and swap. Exchanges the values of *value* and *memory* if the contents of *memory* equals *compare*. Generates the `cmpswap.w` instruction. Only supported for TriCore1.6.x (**--core=tc1.6.x**) and TriCore1.6.2 (**--core=tc1.6.2**). Note that *memory* must be word-aligned.

CRC generate

Intrinsic Function	Description

unsigned int __crc32(unsigned int b, unsigned int a);	Generate the CRC32 checksum of `a` and inverse of `b` and returns the result. Generates the `crc32` instruction. Only supported for TriCore1.6.x (**--core=tc1.6.x**) and TriCore1.6.2 (**--core=tc1.6.2**). For example:
	
	`ld.w    d15,b`
	`ld.w    d0,a`
	`crc32   d2,d15,d0`

Wait for asynchronous event

Intrinsic Function	Description
void volatile __wait(void);	The processor suspends execution until the next enabled interrupt or asynchronous trap event is detected. Generates the `wait` instruction. Only supported for TriCore1.6.x (**--core=tc1.6.x**) and TriCore1.6.2 (**--core=tc1.6.2**).

Initialize circular pointer

Intrinsic Function	Description
__circ void * __initcirc(void * *buf*, unsigned short *bufsize*, unsigned short *byteindex*);	Initialize a circular pointer with a dynamically allocated buffer at run-time. See also Section 1.3.1, *Circular Buffers: __circ*.

Rotate left/right

Intrinsic Function	Description
unsigned int __rol(unsigned int *operand*, unsigned in *count*)	Rotate *operand* left *count* times. The bits that are shifted out are inserted at the right side (bit 31 is shifted to bit 0).
unsigned int __rol(unsigned int *operand*, unsigned in *count*)	Rotate *operand* right *count* times. The bits that are shifted out are inserted at the left side (bit 0 is shifted to bit 31).

Intrinsics used by compiler/libraries

Intrinsic Function	Description
void * volatile __alloc(__size_t *size*);	Allocate memory. Returns a pointer to memory of *size* bytes length. Returns NULL if there is not enough space left. This function is used internally for variable length arrays, it is not to be used by end users.
void * __dotdotdot__(void);	Variable argument '...' operator. Used in library function `va_start()`. Returns the stack offset to the variable argument list.
volatile void __free(void *p);	Deallocates the memory pointed to by `p`. `p` must point to memory earlier allocated by a call to `__alloc()`.

Intrinsic Function	Description
__codeptr __get_return_address(void);	Used by the compiler for profiling when you compile with the option **--profile**. Returns the return address of a function.

1.12. Compiler Generated Sections

The compiler generates code and data in several types of sections. By default the C compiler generates sections with the following names:

```
section_type_prefix[.core_association].module_name.symbol_name
```

A core association, `share`, `private0`, `private1`, `private2` or `clone`, is only present when this is specified for the TriCore 1.6.x or 1.6.2. See Section 1.4, *Multi-Core Support*.

For interrupt vectors and trap vectors the C compiler generates special section names, where the number 0, 1 or 2 refers to a specific core:

```
.text[.inttab{0|1|2}].intvec.vector_number
```

```
.text[.traptab{0|1|2}].trapvec.vector_number
```

When you use a section renaming pragma, the compiler uses the following section naming convention:

```
section_type_prefix[.core_association][.module_name][.symbol_name][.pragma_value]
```

The prefix depends on the type of the section and determines if the section is initialized, constant or uninitialized and which addressing mode is used. The *symbol_name* is either the name of an object or the name of a function.

The following table lists the section types and name prefixes.

Section type	Name prefix	Description
code	.text	program code
neardata	.zdata	initialized __near data
fardata	.data	initialized __far data
nearrom	.zrodata	constant __near data
farrom	.rodata	constant __far data
nearbss	.zbss	uninitialized __near data (cleared)
farbss	.bss	uninitialized __far data (cleared)
nearnoclear	.zbss	uninitialized __near data
farnoclear	.bss	uninitialized __far data
a0data	.data_a0	initialized __a0 data
a0rom	.rodata_a0	constant __a0 data
a0bss	.bss_a0	uninitialized __a0 data (cleared)

Section type	Name prefix	Description
a1data	.data_a1	initialized __a1 data
a1rom	.rodata_a1	constant __a1 data
a1bss	.bss_a1	uninitialized __a1 data (cleared)
a8data	.data_a8	initialized __a8 data
a8rom	.rodata_a8	constant __a8 data
a8bss	.bss_a8	uninitialized __a8 data (cleared)
a9data	.data_a9	initialized __a9 data
a9rom	.rodata_a9	constant __a9 data
a9bss	.bss_a9	uninitialized __a9 data (cleared)

1.12.1. Rename Sections

You can change the default section names with one of the following pragmas. The naming convention for the renamed section is:

```
section_type_prefix[.core_association][.module_name][.symbol_name][.pragma_value]
```

Note however that a symbol at an absolute address (`__at`) is located in a section that always uses the default section name.

#pragma section *type* "*name*"

With this pragma all sections of the specified *type* will be named "*prefix.name*". For example,

```
#pragma section neardata "where"
```

all sections of type `neardata` have the name ".zdata.where".

`#pragma section` *type* will restore the default section naming for sections of this type.

`#pragma section` *type* `restore` will restore the previous setting of `#pragma section` *type*.

When *name* contains exactly one space, the new section name equals the *section_type_prefix*. For example, after `#pragma section code " "` the section name will be `.text`.

#pragma section all "*name*"

With this pragma all sections will be named "*prefix.name*", unless you use a type specific renaming pragma. For example,

```
#pragma section all "here"
```

all sections have the syntax "*prefix*`.here`". For example, sections of type neardata have the name "`.zdata.here`"

`#pragma section all` will restore the default section naming (not for sections that have a type specific renaming pragma).

`#pragma section all restore` will restore the previous setting of `#pragma section all`.

On the command line you can use the C compiler option **--rename-sections**[**=***name*].

Note that when you use one of the above section renaming pragmas, the module name and symbol name are no longer part of the section name. Use one or both of the following pragmas to influence the section naming convention.

#pragma section_name_with_module

With this pragma all section renaming pragmas will use a renaming scheme like:

section_type_prefix.module_name.pragma_value

#pragma section_name_with_symbol

With this pragma all section renaming pragmas will use a renaming scheme like:

section_type_prefix.symbol_name.pragma_value

See also C compiler option **--section-name-with-symbol**.

Examples

```
#pragma section all "rename_1"
 //  .text.rename_1
 //  .data.rename_1

#pragma section code "rename_2"
 //  .text.rename_2
 //  .data.rename_1
```

1.12.2. Influence Section Definition

The following pragmas also influence the section definition:

#pragma section code_init

Code sections are copied from ROM to RAM at program startup.

#pragma section const_init

Sections with constant data are copied from ROM to RAM at program startup.

#pragma section vector_init

Sections with interrupts and trap vectors are copied from ROM to RAM at program startup.

#pragma section data_overlay

The `nearnoclear` and `farnoclear` sections can be overlaid by other sections with the same name. Since by default section naming never leads to sections with the same name, you must force the same name by using one of the section renaming pragmas. To get noclear sections instead of BSS sections you must also use `#pragma noclear`.

Chapter 2. C++ Language

The TASKING C++ compiler (**cptc**) offers a new approach to high-level language programming for the TriCore family. The C++ compiler accepts the C++ language as defined by the ISO/IEC 14882:2003 standard. It also accepts the language extensions of the C compiler (see Chapter 1, *C Language*).

This chapter describes the C++ language implementation and some specific features.

Note that the C++ language itself is not described in this document. For more information on the C++ language, see

* The C++ Programming Language (second edition) by Bjarne Straustrup (1991, Addison Wesley)

* ISO/IEC 14882:1998 C++ standard [ANSI]

* ISO/IEC 14882:2003 C++ standard [ISO/IEC]

* ISO/IEC 14882:2011 C++ standard [ISO/IEC]

* ISO/IEC 14882:2014 C++ standard [ISO/IEC]

  More information on the standards can be found at http://www.iso.org/

2.1. C++ Language Extension Keywords

The C++ compiler supports the same language extension keywords as the C compiler. When option **--strict** is used, the extensions will be disabled.

pragmas

The C++ compiler supports the pragmas as explained in Section 2.7, *Pragmas to Control the C++ Compiler*. Pragmas give directions to the code generator of the compiler.

2.2. C++ Dialect Accepted

The C++ compiler accepts the complete C++ language as defined by the ISO/IEC 14882:2003 standard.

With option **--c++11** the C++ compiler accepts the complete C++11 language as defined by the ISO/IEC 14882:2011 standard.

With option **--c++14** the C++ compiler accepts the complete C++14 language as defined by the ISO/IEC 14882:2014 standard.

Command line options are also available to enable and disable anachronisms and strict standard-conformance checking.

2.2.1. Default C++ Mode

The following extensions are accepted in default C++ mode. Most of these are also accepted in any other C++ mode (except when strict ANSI/ISO violations are diagnosed as errors or were explicitly noted):

- A `friend` declaration for a class may omit the `class` keyword:

```
class A {
    friend B; // Should be "friend class B"
};
```

- Constants of scalar type may be defined within classes:

```
class A {
    const int size = 10;
    int a[size];
};
```

- In the declaration of a class member, a qualified name may be used:

```
struct A {
    int A::f(); // Should be int f();
};
```

- The `restrict` keyword is allowed.

- A `const` qualified object with file scope or namespace scope and the `__at()` attribute will have external linkage, unless explicitly declared `static`. Examples:

```
const int i = 5;                        // internal linkage
const int j __at( 0x1234 ) = 10;        // external linkage
static const int k __at( 0x1236 ) = 15;  // internal linkage
```

Note that no warning is generated when 'j' is not used.

- Implicit type conversion between a pointer to an `extern "C"` function and a pointer to an `extern "C++"` function is permitted. Here's an example:

```
extern "C" void f(); // f's type has extern "C" linkage
void (*pf)() // pf points to an extern "C++" function
    = &f;    // error unless implicit conversion is
             // allowed
```

This extension is allowed in environments where C and C++ functions share the same calling conventions. It is enabled by default.

- A "**?**" operator whose second and third operands are string literals or wide string literals can be implicitly converted to "`char *`" or "`wchar_t *`". (Recall that in C++ string literals are `const`. There is a deprecated implicit conversion that allows conversion of a string literal to "`char *`", dropping the `const`.

That conversion, however, applies only to simple string literals. Allowing it for the result of a "**?**" operation is an extension.)

```
char *p = x ? "abc" : "def";
```

- Default arguments may be specified for function parameters other than those of a top-level function declaration (e.g., they are accepted on `typedef` declarations and on pointer-to-function and pointer-to-member-function declarations).

- Non-static local variables of an enclosing function can be referenced in a non-evaluated expression (e.g., a `sizeof` expression) inside a local class. A warning is issued.

- In default C++ mode, the friend class syntax is extended to allow nonclass types as well as class types expressed through a typedef or without an elaborated type name. For example:

```
typedef struct S ST;
class C {
    friend S;       // OK (requires S to be in scope).
    friend ST;      // OK (same as "friend S;").
    friend int;     // OK (no effect).
    friend S const; // Error: cv-qualifiers cannot
                    // appear directly.
};
```

- In default C++ mode, mixed string literal concatenations are accepted. (This is a feature carried over from C99 and also available in GNU modes).

```
wchar_t *str = "a" L"b"; // OK, same as L"ab".
```

- In default C++ mode, variadic macros are accepted. (This is a feature carried over from C99 and also available in GNU modes.)

- In default C++ mode, empty macro arguments are accepted (a feature carried over from C99).

- A trailing comma in the definition of an enumeration type is silently accepted (a feature carried over from C99):

```
enum E { e, };
```

2.2.2. GNU C++ Mode

The C++ compiler can be configured to support GNU C++ (command line option **--g++**). In GNU C++ mode, many extensions provided by the GNU C++ compiler are accepted.

Because the GNU C++ compiler frequently changes behavior between releases, the C++ compiler provides an option (**--gnu_version**) to specify a specific version of GCC to emulate. Generally speaking, features and bugs are emulated to exactly match each known version of GCC, but occasionally the emulation is approximate and in such cases the C++ compiler is often a little more permissive than GCC on the principle that it is more important to accept source that GCC accepts than to diagnose every case that GCC

diagnoses. The C++ compiler does not, however, attempt to emulate every GCC command line option; in particular, GCC options to be extra-permissive are not emulated (however, the severity of specific error diagnostics can sometimes be decreased to accept constructs that are not by-default allowed in GNU emulation mode).

The following GNU extensions are *not* supported:

- The forward declaration of function parameters (so they can participate in variable-length array parameters).

- GNU-style complex integral types (complex floating-point types are supported)

- Nested functions

- Local structs with variable-length array fields.

2.2.3. Anachronisms Accepted

The following anachronisms are accepted when anachronisms are enabled (with **--anachronisms**):

- `overload` is allowed in function declarations. It is accepted and ignored.

- Definitions are not required for static data members that can be initialized using default initialization. The anachronism does not apply to static data members of template classes; they must always be defined.

- The number of elements in an array may be specified in an array `delete` operation. The value is ignored.

- A single `operator++()` and `operator--()` function can be used to overload both prefix and postfix operations.

- The base class name may be omitted in a base class initializer if there is only one immediate base class.

- Assignment to `this` in constructors and destructors is allowed. This is allowed only if anachronisms are enabled and the "assignment to `this`" configuration parameter is enabled.

- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.

- A nested class name may be used as a non-nested class name provided no other class of that name has been declared. The anachronism is not applied to template classes.

- A reference to a non-const type may be initialized from a value of a different type. A temporary is created, it is initialized from the (converted) initial value, and the reference is set to the temporary.

- A reference to a non-const class type may be initialized from an rvalue of the class type or a derived class thereof. No (additional) temporary is used.

- A function with old-style parameter declarations is allowed and may participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such

functions when the check for compatibility is done, so that the following declares the overloading of two functions named `f`:

```
int f(int);
int f(x) char x; { return x; }
```

Note that in C this code is legal but has a different meaning: a tentative declaration of `f` is followed by its definition.

• When option **--nonconst-ref-anachronism** is set, a reference to a non-const class can be bound to a class rvalue of the same type or a derived type thereof.

```
struct A {
    A(int);
    A operator=(A&);
    A operator+(const A&);
};
main () {
    A b(1);
    b = A(1) + A(2); // Allowed as anachronism
}
```

2.2.4. Atomic Operations

In C++11 mode (option **--c++11**) the C++ compiler supports lock-free atomic operations in machine word size.

The atomic operations are implemented through the use of atomic operations provided by the TriCore instruction set architecture. The C++ compiler does not use mutexes or other locking mechanisms to stall threads. The implementation is limited to atomic operations on variables of type `int` only.

See the header file "*installation_directory*/ctc/include.cpp/atomic" for an overview of the atomic operations that you can use.

Example (`atomic.cpp`)

```
#include <atomic>
#include <iostream>

volatile int data_buffer;
std::atomic_int atom( 1 ); // This variable is volatile by type definition

void thread_on_core1()
{
    data_buffer = 2;
    ++atom; // Making atom 2 informs thread_on_core2 that data_buffer is
            // valid. The ++ uses the atomic cmpswap instruction
    while ( atom != 3 )
    { /* empty */ }  // Wait until atom becomes 3,
                     // set by thread 2, meaning data_buffer is updated
```

```
    std::cout <<data_buffer << std::endl;
              // If data_buffer is not volatile, data_buffer would still be 2
}
void thread_on_core2()
{
    while ( atom != 2 )
    { /* empty */ }  // Wait until atom becomes 2,
                      // which means that data_buffer can be used
    std::cout << data_buffer << std::endl;
    data_buffer = 3; // Set data_buffer to 3 and
    ++atom;          // Inform the other thread that data_buffer
                     // can be used with new value
}

int main(void)
{
    thread_on_core1();
    thread_on_core2();
    return 0;
}
```

Use the following command to build the example:

```
cctc --c++11 --core=tc1.6.x --io-streams atomic.cpp
```

Use the options **--c++11** and **--core=tc1.6.x** to enable the use of atomic operations.

Objects of atomic type and the data objects that are protected by atomic types must be allocated in non-cached memory. You can do this either by disabling the caches or by updating the linker script language file (LSL file). In the latter case the sections in which the objects are allocated must be selected and allocated at addresses that correspond to non-cached memory.

The following LSL part shows how to locate the section .zdata.atomic.atom in the not_cached area of lmuram.

```
#include <cpu.lsl>

section_layout mpe:vtc:abs18
{
    group not_cached ( ordered, run_addr=mem:mpe:lmuram/not_cached )
    {
        select ".zdata.atomic.atom";
    }
}
```

2.3. Namespace Support

Namespaces are enabled by default. You can use the command line option **--no-namespaces** to disable the features.

When doing name lookup in a template instantiation, some names must be found in the context of the template definition while others may also be found in the context of the template instantiation. The C++ compiler implements two different instantiation lookup algorithms: the one mandated by the standard (referred to as "dependent name lookup"), and the one that existed before dependent name lookup was implemented.

Dependent name lookup is done in strict mode (unless explicitly disabled by another command line option) or when dependent name processing is enabled by either a configuration flag or command line option.

Dependent Name Processing

When doing dependent name lookup, the C++ compiler implements the instantiation name lookup rules specified in the standard. This processing requires that non-class prototype instantiations be done. This in turn requires that the code be written using the typename and template keywords as required by the standard.

Lookup Using the Referencing Context

When not using dependent name lookup, the C++ compiler uses a name lookup algorithm that approximates the two-phase lookup rule of the standard, but does so in such a way that is more compatible with existing code and existing compilers.

When a name is looked up as part of a template instantiation but is not found in the local context of the instantiation, it is looked up in a synthesized instantiation context that includes both names from the context of the template definition and names from the context of the instantiation. Here's an example:

```
namespace N {
    int g(int);
    int x = 0;
    template <class T> struct A {
        T f(T t) { return g(t); }
        T f() { return x; }
    };
}

namespace M {
    int x = 99;
    double g(double);
    N::A<int> ai;
    int i = ai.f(0);    // N::A<int>::f(int) calls
                        // N::g(int)
    int i2 = ai.f();    // N::A<int>::f() returns
                        // 0 (= N::x)
    N::A<double> ad;
    double d = ad.f(0); // N::A<double>::f(double)
```

```
                            // calls M::g(double)
    double d2 = ad.f(); // N::A<double>::f() also
                        // returns 0 (= N::x)
}
```

The lookup of names in template instantiations does not conform to the rules in the standard in the following respects:

- Although only names from the template definition context are considered for names that are not functions, the lookup is not limited to those names visible at the point at which the template was defined.

- Functions from the context in which the template was referenced are considered for all function calls in the template. Functions from the referencing context should only be visible for "dependent" function calls.

Argument Dependent Lookup

When argument-dependent lookup is enabled (this is the default), functions made visible using argument-dependent lookup overload with those made visible by normal lookup. The standard requires that this overloading occurs even when the name found by normal lookup is a block `extern` declaration. The C++ compiler does this overloading, but in default mode, argument-dependent lookup is suppressed when the normal lookup finds a block `extern`.

This means a program can have different behavior, depending on whether it is compiled with or without argument-dependent lookup **--no-arg-dep-lookup**, even if the program makes no use of namespaces. For example:

```
struct A { };
A operator+(A, double);
void f() {
    A a1;
    A operator+(A, int);
    a1 + 1.0; // calls operator+(A, double)
              // with arg-dependent lookup enabled but
              // otherwise calls operator+(A, int);
}
```

2.4. Template Instantiation

The C++ language includes the concept of *templates*. A template is a description of a class or function that is a model for a family of related classes or functions.[1] For example, one can write a template for a `Stack` class, and then use a stack of integers, a stack of floats, and a stack of some user-defined type. In the source, these might be written `Stack<int>`, `Stack<float>`, and `Stack<X>`. From a single source description of the template for a stack, the compiler can create *instantiations* of the template for each of the types required.

---

[1]Since templates are descriptions of entities (typically, classes) that are parameterizable according to the types they operate upon, they are sometimes called *parameterized types*.

The instantiation of a class template is always done as soon as it is needed in a compilation. However, the instantiations of template functions, member functions of template classes, and static data members of template classes (hereafter referred to as template entities) are not necessarily done immediately, for several reasons:

- One would like to end up with only one copy of each instantiated entity across all the object files that make up a program. (This of course applies to entities with external linkage.)

- The language allows one to write a *specialization* of a template entity, i.e., a specific version to be used in place of a version generated from the template for a specific data type. (One could, for example, write a version of Stack<int>, or of just Stack<int>::push, that replaces the template-generated version; often, such a specialization provides a more efficient representation for a particular data type.) Since the compiler cannot know, when compiling a reference to a template entity, if a specialization for that entity will be provided in another compilation, it cannot do the instantiation automatically in any source file that references it.

- The language also dictates that template functions that are not referenced should not be compiled, that, in fact, such functions might contain semantic errors that would prevent them from being compiled. Therefore, a reference to a template class should not automatically instantiate all the member functions of that class.

(It should be noted that certain template entities are always instantiated when used, e.g., inline functions.)

From these requirements, one can see that if the compiler is responsible for doing all the instantiations automatically, it can only do so on a program-wide basis. That is, the compiler cannot make decisions about instantiation of template entities until it has seen all the source files that make up a complete program.

This C++ compiler provides an instantiation mechanism that does automatic instantiation at link time. For cases where you want more explicit control over instantiation, the C++ compiler also provides instantiation modes and instantiation pragmas, which can be used to exert fine-grained control over the instantiation process.

2.4.1. Instantiation Modes

Normally, when a file is compiled, template entities are instantiated everywhere where they are used. The overall instantiation mode can, however, be changed by a command line option:

--instantiate=used

Instantiate those template entities that were used in the compilation. This will include all static data members for which there are template definitions. This is the default.

--instantiate=all

Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members will be instantiated whether or not they were used. Non-member template functions will be instantiated even if the only reference was a declaration.

--instantiate=local

Similar to **--instantiate=used** except that the functions are given internal linkage. This is intended to provide a very simple mechanism for those getting started with templates. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly (barring problems due to multiple copies of local static variables.) However, one may end up with many copies of the instantiated functions, so this is not suitable for production use. **--instantiate=local** cannot be used in conjunction with automatic template instantiation. If automatic instantiation is enabled by default, it will be disabled by the **--instantiate=local** option.

In the case where the **cctc** command is given a single file to compile and link, e.g.,

```
cctc test.cc
```

the compiler knows that all instantiations will have to be done in the single source file. Therefore, it uses the **--instantiate=used** mode and suppresses automatic instantiation.

2.4.2. Instantiation #pragma Directives

Instantiation pragmas can be used to control the instantiation of specific template entities or sets of template entities. There are three instantiation pragmas:

- The **instantiate** pragma causes a specified entity to be instantiated.

- The **do_not_instantiate** pragma suppresses the instantiation of a specified entity. It is typically used to suppress the instantiation of an entity for which a specific definition will be supplied.

- The **can_instantiate** pragma indicates that a specified entity can be instantiated in the current compilation, but need not be; it is used in conjunction with automatic instantiation, to indicate potential sites for instantiation if the template entity turns out to be required.

The argument to the instantiation pragma may be:

- a template class name `A<int>`

- a template class declaration `class A<int>`

- a member function name `A<int>::f`

- a static data member name `A<int>::i`

- a static data declaration `int A<int>::i`

- a member function declaration `void A<int>::f(int,char)`

- a template function declaration `char* f(int, float)`

A pragma in which the argument is a template class name (e.g., `A<int>` or `class A<int>`) is equivalent to repeating the pragma for each member function and static data member declared in the class. When instantiating an entire class a given member function or static data member may be excluded using the **do_not_instantiate** pragma. For example,

```
#pragma instantiate A<int>
#pragma do_not_instantiate A<int>::f
```

The template definition of a template entity must be present in the compilation for an instantiation to occur. If an instantiation is explicitly requested by use of the **instantiate** pragma and no template definition is available or a specific definition is provided, an error is issued.

```
template <class T> void f1(T);  // No body provided
template <class T> void g1(T);  // No body provided

void f1(int) {} // Specific definition
void main()
{
    int i;
    double d;
    f1(i);
    f1(d);
    g1(i);
    g1(d);
}

#pragma instantiate void f1(int) // error - specific
                                 // definition
#pragma instantiate void g1(int) // error - no body
                                 // provided
```

`f1(double)` and `g1(double)` will not be instantiated (because no bodies were supplied) but no errors will be produced during the compilation (if no bodies are supplied at link time, a linker error will be produced).

A member function name (e.g., `A<int>::f`) can only be used as a pragma argument if it refers to a single user defined member function (i.e., not an overloaded function). Compiler-generated functions are not considered, so a name may refer to a user defined constructor even if a compiler-generated copy constructor of the same name exists. Overloaded member functions can be instantiated by providing the complete member function declaration, as in

```
#pragma instantiate char* A<int>::f(int, char*)
```

The argument to an instantiation pragma may not be a compiler-generated function, an inline function, or a pure virtual function.

2.4.3. Implicit Inclusion

When implicit inclusion is enabled, the C++ compiler is given permission to assume that if it needs a definition to instantiate a template entity declared in a `.h` file it can implicitly include the corresponding `.cc` file to get the source code for the definition. For example, if a template entity `ABC::f` is declared in file `xyz.h`, and an instantiation of `ABC::f` is required in a compilation but no definition of `ABC::f` appears in the source code processed by the compilation, the compiler will look to see if a file `xyz.cc` exists, and if so it will process it as if it were included at the end of the main source file.

To find the template definition file for a given template entity the C++ compiler needs to know the path name specified in the original include of the file in which the template was declared and whether the file was included using the system include syntax (e.g., `#include <file.h>`). This information is not available for preprocessed source containing `#line` directives. Consequently, the C++ compiler will not attempt implicit inclusion for source code containing `#line` directives.

The file to be implicitly included is found by replacing the file suffix with each of the suffixes specified in the instantiation file suffix list. The normal include search path mechanism is then used to look for the file to be implicitly included.

By default, the list of definition file suffixes tried is `.c`, `.cc`, `.cpp`, and `.cxx`.

Implicit inclusion works well alongside automatic instantiation, but the two are independent. They can be enabled or disabled independently, and implicit inclusion is still useful when automatic instantiation is not done.

The implicit inclusion mode can be turned on by the command line option **--implicit-include**.

Implicit inclusions are only performed during the normal compilation of a file, (i.e., not when doing only preprocessing). A common means of investigating certain kinds of problems is to produce a preprocessed source file that can be inspected. When using implicit inclusion it is sometimes desirable for the preprocessed source file to include any implicitly included files. This may be done using the command line option **--no-preprocessing-only**. This causes the preprocessed output to be generated as part of a normal compilation. When implicit inclusion is being used, the implicitly included files will appear as part of the preprocessed output in the precise location at which they were included in the compilation.

2.5. Inlining Functions

The C++ compiler supports a minimal form of function inlining. When the C++ compiler encounters a call of a function declared `inline` it can replace the call with the body of the function with the parameters replaced by the corresponding arguments. When a function call occurs as a statement, the statements of the function body are inserted in place of the call. When the function call occurs within an expression, the body of the function is rewritten as one large expression and that expression is inserted in the proper place in the containing expression. It is not always possible to do this sort of inlining: there are certain constructs (e.g. loops and inline assembly) that cannot be rendered in expression form. Even when inlining is done at the statement level, there are certain constructs that are not practical to inline. Calls that cannot be inlined are left in their original call form, and an out-of-line copy of the function is used. When enabled, a remark is issued.

When the C++ compiler decides not to inline a function, the keyword `inline` is passed to the generated C file. This allows for the C compiler to decide again whether to inline a function or not.

A function is disqualified for inlining immediately if any of the following are true:

• The function has local static variables.

• The function has local constants.

• The function has local types.

• The function has block scopes.

- The function includes pragmas.

- The function has a variable argument list.

2.6. Extern Inline Functions

Depending on the way in which the C++ compiler is configured, out-of-line copies of `extern inline` functions are either implemented using static functions, or are instantiated using a mechanism like the template instantiation mechanism. Note that out-of-line copies of inline functions are only required in cases where the function cannot be inlined, or when the address of the function is taken (whether explicitly by the user, by implicitly generated functions, or by compiler-generated data structures such as virtual function tables or exception handling tables).

When static functions are used, local static variables of the functions are promoted to global variables with specially encoded names, so that even though there may be multiple copies of the code, there is only one copy of such global variables. This mechanism does not strictly conform to the standard because the address of an extern inline function is not constant across translation units.

When the instantiation mechanism is used, the address of an extern inline function is constant across translation units, but at the cost of requiring the use of one of the template instantiation mechanisms, even for programs that don't use templates. Definitions of extern inline functions can be provided either through use of the automatic instantiation mechanism or by use of the **--instantiate=used** or **--instantiate=all** instantiation modes. There is no mechanism to manually control the definition of extern inline function bodies.

2.7. Pragmas to Control the C++ Compiler

Pragmas are keywords in the C++ source that control the behavior of the compiler. Pragmas overrule compiler options.

The syntax is:

```
#pragma pragma-spec
```

The C++ compiler supports the following pragmas:

instantiate / do_not_instantiate / can_instantiate

These are template instantiation pragmas. They are described in detail in Section 2.4.2, *Instantiation #pragma Directives*.

hdrstop / no_pch

These are precompiled header pragmas. They are described in detail in Section 2.9, *Precompiled Headers*.

once

When placed at the beginning of a header file, indicates that the file is written in such a way that including it several times has the same effect as including it once. Thus, if the C++ compiler sees `#pragma once` at the start of a header file, it will skip over it if the file is #included again.

A typical idiom is to place an #ifndef guard around the body of the file, with a #define of the guard variable after the #ifndef:

```
#pragma once    // optional
#ifndef FILE_H
#define FILE_H
... body of the header file ...
#endif
```

The `#pragma once` is marked as optional in this example, because the C++ compiler recognizes the `#ifndef` idiom and does the optimization even in its absence. `#pragma once` is accepted for compatibility with other compilers and to allow the programmer to use other guard-code idioms.

2.7.1. C pragmas Supported by the C++ compiler

The C++ compiler supports the following C pragmas with the restriction that the `#pragma` can only be used if it is placed in C++ code adjacent to the variable or function it is meant for:

clear / noclear

By default, uninitialized global or static variables are cleared to zero on startup. With pragma `noclear`, this step is skipped. Pragma `clear` resumes normal behavior. This pragma applies to constant data as well as non-constant data. It is mandatory to use `#pragma clear`/`#pragma noclear` directly before the variable and `#pragma noclear`/`#pragma clear` directly after the variable. Unlike in C, these pragmas do not allow arguments.

```
#pragma noclear    // adjacent to variable
int var;
#pragma clear      // adjacent to variable
```

message "*message*" ...

Print the message string(s) on standard output.

section all "*name*"
section *type* "*name*"
section_name_with_module
section_name_with_symbol

Changes section names. See Section 1.12, *Compiler Generated Sections* and C compiler option **--rename-sections** for more information.

section code_init | const_init | vector_init

At startup copies corresponding sections to RAM for initialization. See Section 1.12.2, *Influence Section Definition*.

section data_overlay

Allow overlaying data sections.

2.8. Predefined Macros

The C++ compiler defines a number of preprocessing macros. Many of them are only defined under certain circumstances. This section describes the macros that are provided and the circumstances under which they are defined.

Macro	Description
__ABI_COMPATIBILITY_VERSION	Defines the ABI compatibility version being used. This macro is set to 9999, which means the latest version. This macro is used when building the C++ library.
__ABI_CHANGES_FOR_RTTI	This macro is set to TRUE, meaning that the ABI changes for RTTI are implemented. This macro is used when building the C++ library.
__ABI_CHANGES_FOR_ARRAY_NEW_AND_DELETE	This macro is set to TRUE, meaning that the ABI changes for array new and delete are implemented. This macro is used when building the C++ library.
__ABI_CHANGES_FOR_PLACEMENT_DELETE	This macro is set to TRUE, meaning that the ABI changes for placement delete are implemented. This macro is used when building the C++ library.
__ARRAY_OPERATORS	Defined when array `new` and `delete` are enabled. This is the default.
__BASE_FILE__	Similar to __FILE__ but indicates the primary source file rather than the current one (i.e., when the current file is an included file).
_BOOL	Defined when `bool` is a keyword. This is the default.

Macro	Description
__BUILD__	Identifies the build number of the C++ compiler, composed of decimal digits for the build number, three digits for the major branch number and three digits for the minor branch number. For example, if you use build 1.22.1 of the compiler, __BUILD__ expands to 1022001. If there is no branch number, the branch digits expand to zero. For example, build 127 results in 127000000.
__CHAR_MIN / __CHAR_MAX	Used in limits.h to define the minimum/maximum value of a plain char respectively.
__CHAR16_T_AND_CHAR32_T	Defined when char16_t and char32_t are keywords. These keywords are enabled when you use C++ compiler option **--uliterals**.
__CHAR16_TYPE__	Defined as the underlying type for char16_t (GNU version 40400 and above, see **--gnu-version**).
__CHAR32_TYPE__	Defined as the underlying type for char32_t (GNU version 40400 and above, see **--gnu-version**).
__CORE_*core*__	A symbol is defined depending on the option **--core=***core*. The *core* is converted to uppercase and '.' is removed. For example, if **--core=tc1.3.1** is specified, the symbol __CORE_TC131__ is defined. When no **--core** is supplied, the C++ compiler defines __CORE_TC13__.
__CPTC__	Identifies the C++ compiler. You can use this symbol to flag parts of the source which must be recognized by the **cptc** C++ compiler only. It expands to 1.
__cplusplus	Always defined.
__CPU__	Expands to the name of the CPU supplied with the control program option **--cpu=***cpu*. When no **--cpu** is supplied, this symbol is not defined. For example, if **--cpu=tc1796b** is specified to the control program, the symbol __CPU__ expands to tc1796b.
__DATE__	Defined to the date of the compilation in the form "Mmm dd yyyy".
__DELTA_TYPE	Defines the type of the offset field in the virtual function table. This macro is used when building the C++ library.

Macro	Description
__DOUBLE_FP__	Expands to 1 if you did *not* use option **--no-double** (Treat 'double' as 'float'), otherwise unrecognized as macro.
__embedded_cplusplus	Defined as 1 in Embedded C++ mode.
__EXCEPTIONS	Defined when exception handling is enabled (**--exceptions**).
__FILE__	Expands to the current source file name.
__FUNCTION__	Defined to the name of the current function. An error is issued if it is used outside of a function.
__func__	Same as __FUNCTION__ in GNU mode.
__IMPLICIT_USING_STD	Defined when the standard header files should implicitly do a using-directive on the std namespace (**--using-std**).
__JMP_BUF_ELEMENT_TYPE	Specifies the type of an element of the setjmp buffer. This macro is used when building the C++ library.
__JMP_BUF_NUM_ELEMENTS	Defines the number of elements in the setjmp buffer. This macro is used when building the C++ library.
__LINE__	Expands to the line number of the line where this macro is called.
__NAMESPACES	Defined when namespaces are supported (this is the default, you can disable support for namespaces with **--no-namespaces**).
__NO_LONG_LONG	Defined when the long long type is not supported. This is the default.
__NULL_EH_REGION_NUMBER	Defines the value used as the null region number value in the exception handling tables. This macro is used when building the C++ library.
__PLACEMENT_DELETE	Defined when placement delete is enabled.
__PRETTY_FUNCTION__	Defined to the name of the current function. This includes the return type and parameter types of the function. An error is issued if it is used outside of a function.
__PTRDIFF_MIN / __PTRDIFF_MAX	Used in stdint.h to define the minimum/maximum value of a ptrdiff_t type respectively.
__PTRDIFF_TYPE__	Defined to be the type of ptrdiff_t.

Macro	Description
__REGION_NUMBER_TYPE	Defines the type of a region number field in the exception handling tables. This macro is used when building the C++ library.
__REVISION__	Expands to the revision number of the C++ compiler. Digits are represented as they are; characters (for prototypes, alphas, betas) are represented by -1. Examples: v1.0r1 -> 1, v1.0rb -> -1
__RTTI	Defined when RTTI is enabled (**--rtti**).
__RUNTIME_USES_NAMESPACES	Defined when the run-time uses namespaces.
__SFRFILE__(*cpu*)	If control program option **--cpu=***cpu* is specified, this macro expands to the filename of the used SFR file, including the pathname and the < >. The *cpu* is the argument of the macro. For example, if **--cpu=tc1796b** is specified, the macro `__SFRFILE__(__CPU__)` expands to `__SFRFILE__(tc1796b)`, which expands to `<sfr/regtc1796b.sfr>`.
__SIGNED_CHARS__	Defined when plain `char` is signed.
__SINGLE_FP__	Expands to 1 if you used option **--no-double** (Treat 'double' as 'float'), otherwise unrecognized as macro.
__SIZE_MIN / __SIZE_MAX	Used in `stdint.h` to define the minimum/maximum value of a `size_t` type respectively.
__SIZE_TYPE__	Defined to be the type of `size_t`.
__STDC__	Always defined, but the value may be redefined.
__STDC_VERSION__	Identifies the ISO-C version number. Expands to 199901L for ISO C99, but the value may be redefined.
_STLP_NO_IOSTREAMS	Defined when option **--io-streams** is not used. This disables I/O stream functions in the STLport C++ library.
__TASKING__	Always defined for the TASKING C++ compiler.
__TIME__	Expands to the compilation time: "hh:mm:ss"
__TYPE_TRAITS_ENABLED	Defined when type traits pseudo-functions (to ease the implementation of ISO/IEC TR 19768; e.g., `__is_union`) are enabled. This is the default in C++ mode.

Macro	Description
__VAR_HANDLE_TYPE	Defines the type of the variable-handle field in the exception handling tables. This macro is used when building the C++ library.
__VARIADIC_TEMPLATES	Defined when C++11 variadic templates are supported (option **--variadic-templates**).
__VERSION__	Identifies the version number of the C++ compiler. For example, if you use version 2.1r1 of the compiler, __VERSION__ expands to 2001 (dot and revision number are omitted, minor version number in 3 digits).
__VIRTUAL_FUNCTION_INDEX_TYPE	Defines the type of the virtual function index field of the virtual function table. This macro is used when building the C++ library.
__WCHAR_MIN / __WCHAR_MAX	Used in `stdint.h` to define the minimum/maximum value of a `wchar_t` type respectively.
_WCHAR_T	Defined when `wchar_t` is a keyword.

2.9. Precompiled Headers

It is often desirable to avoid recompiling a set of header files, especially when they introduce many lines of code and the primary source files that `#include` them are relatively small. The C++ compiler provides a mechanism for, in effect, taking a snapshot of the state of the compilation at a particular point and writing it to a disk file before completing the compilation; then, when recompiling the same source file or compiling another file with the same set of header files, it can recognize the "snapshot point", verify that the corresponding precompiled header (PCH) file is reusable, and read it back in. Under the right circumstances, this can produce a dramatic improvement in compilation time; the trade-off is that PCH files can take a lot of disk space.

2.9.1. Automatic Precompiled Header Processing

When **--pch** appears on the command line, automatic precompiled header processing is enabled. This means the C++ compiler will automatically look for a qualifying precompiled header file to read in and/or will create one for use on a subsequent compilation.

The PCH file will contain a snapshot of all the code preceding the "header stop" point. The header stop point is typically the first token in the primary source file that does not belong to a preprocessing directive, but it can also be specified directly by **#pragma hdrstop** (see below) if that comes first. For example:

```
#include "xxx.h"
#include "yyy.h"
int i;
```

The header stop point is `int` (the first non-preprocessor token) and the PCH file will contain a snapshot reflecting the inclusion of `xxx.h` and `yyy.h`. If the first non-preprocessor token or the `#pragma hdrstop`

appears within a `#if` block, the header stop point is the outermost enclosing `#if`. To illustrate, heres a more complicated example:

```
#include "xxx.h"
#ifndef YYY_H
#define YYY_H 1
#include "yyy.h"
#endif
#if TEST
int i;
#endif
```

Here, the first token that does not belong to a preprocessing directive is again `int`, but the header stop point is the start of the `#if` block containing it. The PCH file will reflect the inclusion of `xxx.h` and conditionally the definition of `YYY_H` and inclusion of `yyy.h`; it will not contain the state produced by `#if TEST`.

A PCH file will be produced only if the header stop point and the code preceding it (mainly, the header files themselves) meet certain requirements:

• The header stop point must appear at file scope -- it may not be within an unclosed scope established by a header file. For example, a PCH file will not be created in this case:

```
// xxx.h
class A {

// xxx.C
#include "xxx.h"
int i; };
```

• The header stop point may not be inside a declaration started within a header file, nor (in C++) may it be part of a declaration list of a linkage specification. For example, in the following case the header stop point is int, but since it is not the start of a new declaration, no PCH file will be created:

```
// yyy.h
static

// yyy.C
#include "yyy.h"
int i;
```

• Similarly, the header stop point may not be inside a `#if` block or a `#define` started within a header file.

• The processing preceding the header stop must not have produced any errors. (Note: warnings and other diagnostics will not be reproduced when the PCH file is reused.)

• No references to predefined macros `__DATE__` or `__TIME__` may have appeared.

• No use of the `#line` preprocessing directive may have appeared.

- **#pragma no_pch** (see below) must not have appeared.

- The code preceding the header stop point must have introduced a sufficient number of declarations to justify the overhead associated with precompiled headers. The minimum number of declarations required is 1.

When the host system does not support memory mapping, so that everything to be saved in the precompiled header file is assigned to preallocated memory (MS-Windows), two additional restrictions apply:

- The total memory needed at the header stop point cannot exceed the size of the block of preallocated memory.

- No single program entity saved can exceed 16384, the preallocation unit.

When a precompiled header file is produced, it contains, in addition to the snapshot of the compiler state, some information that can be checked to determine under what circumstances it can be reused. This includes:

- The compiler version, including the date and time the compiler was built.

- The current directory (i.e., the directory in which the compilation is occurring).

- The command line options.

- The initial sequence of preprocessing directives from the primary source file, including #include directives.

- The date and time of the header files specified in #include directives.

This information comprises the PCH prefix. The prefix information of a given source file can be compared to the prefix information of a PCH file to determine whether the latter is applicable to the current compilation.

As an illustration, consider two source files:

```
// a.cc
#include "xxx.h"
 ...              // Start of code
// b.cc
#include "xxx.h"
 ...              // Start of code
```

When a.cc is compiled with **--pch**, a precompiled header file named a.pch is created. Then, when b.cc is compiled (or when a.cc is recompiled), the prefix section of a.pch is read in for comparison with the current source file. If the command line options are identical, if xxx.h has not been modified, and so forth, then, instead of opening xxx.h and processing it line by line, the C++ compiler reads in the rest of a.pch and thereby establishes the state for the rest of the compilation.

It may be that more than one PCH file is applicable to a given compilation. If so, the largest (i.e., the one representing the most preprocessing directives from the primary source file) is used. For instance, consider a primary source file that begins with

```
#include "xxx.h"
#include "yyy.h"
#include "zzz.h"
```

If there is one PCH file for xxx.h and a second for xxx.h *and* yyy.h, the latter will be selected (assuming both are applicable to the current compilation). Moreover, after the PCH file for the first two headers is read in and the third is compiled, a new PCH file for all three headers may be created.

When a precompiled header file is created, it takes the name of the primary source file, with the suffix replaced by an implementation-specified suffix (pch by default). Unless **--pch-dir** is specified (see below), it is created in the directory of the primary source file.

When a precompiled header file is created or used, a message such as

```
"test.cc": creating precompiled header file "test.pch"
```

is issued. The user may suppress the message by using the command line option **--no-pch-messages**.

When the option **--pch-verbose** is used the C++ compiler will display a message for each precompiled header file that is considered that cannot be used giving the reason that it cannot be used.

In automatic mode (i.e., when **--pch** is used) the C++ compiler will deem a precompiled header file obsolete and delete it under the following circumstances:

- if the precompiled header file is based on at least one out-of-date header file but is otherwise applicable for the current compilation; or

- if the precompiled header file has the same base name as the source file being compiled (e.g., xxx.pch and xxx.cc) but is not applicable for the current compilation (e.g., because of different command line options).

This handles some common cases; other PCH file clean-up must be dealt with by other means (e.g., by the user).

Support for precompiled header processing is not available when multiple source files are specified in a single compilation: an error will be issued and the compilation aborted if the command line includes a request for precompiled header processing and specifies more than one primary source file.

2.9.2. Manual Precompiled Header Processing

Command line option **--create-pch=**ment*file-name* specifies that a precompiled header file of the specified name should be created.

Command line option **--use-pch=**ment*file-name* specifies that the indicated precompiled header file should be used for this compilation; if it is invalid (i.e., if its prefix does not match the prefix for the current primary source file), a warning will be issued and the PCH file will not be used.

When either of these options is used in conjunction with **--pch-dir**, the indicated file name (which may be a path name) is tacked on to the directory name, unless the file name is an absolute path name.

The options **--create-pch**, **--use-pch**, and **--pch** may not be used together. If more than one of these options is specified, only the last one will apply. Nevertheless, most of the description of automatic PCH

processing applies to one or the other of these modes -- header stop points are determined the same way, PCH file applicability is determined the same way, and so forth.

2.9.3. Other Ways to Control Precompiled Headers

There are several ways in which the user can control and/or tune how precompiled headers are created and used.

- **#pragma hdrstop** may be inserted in the primary source file at a point prior to the first token that does not belong to a preprocessing directive. It enables you to specify where the set of header files subject to precompilation ends. For example,

```
#include "xxx.h"
#include "yyy.h"
#pragma hdrstop
#include "zzz.h"
```

  Here, the precompiled header file will include processing state for xxx.h and yyy.h but not zzz.h. (This is useful if the user decides that the information added by what follows the **#pragma hdrstop** does not justify the creation of another PCH file.)

- **#pragma no_pch** may be used to suppress precompiled header processing for a given source file.

- Command line option **--pch-dir=*directory-name*** is used to specify the directory in which to search for and/or create a PCH file.

Moreover, when the host system does not support memory mapping and preallocated memory is used instead, then one of the command line options **--pch**, **--create-pch**, or **--use-pch**, if it appears at all, must be the *first* option on the command line.

2.9.4. Performance Issues

The relative overhead incurred in writing out and reading back in a precompiled header file is quite small for reasonably large header files.

In general, it does not cost much to write a precompiled header file out even if it does not end up being used, and if it *is* used it almost always produces a significant speedup in compilation. The problem is that the precompiled header files can be quite large (from a minimum of about 250K bytes to several megabytes or more), and so one probably does not want many of them sitting around.

Thus, despite the faster recompilations, precompiled header processing is not likely to be justified for an arbitrary set of files with nonuniform initial sequences of preprocessing directives. Rather, the greatest benefit occurs when a number of source files can share the same PCH file. The more sharing, the less disk space is consumed. With sharing, the disadvantage of large precompiled header files can be minimized, without giving up the advantage of a significant speedup in compilation times.

Consequently, to take full advantage of header file precompilation, users should expect to reorder the #include sections of their source files and/or to group #include directives within a commonly used header file.

Below is an example of how this can be done. A common idiom is this:

```
#include "comnfile.h"
#pragma hdrstop
#include ...
```

where `comnfile.h` pulls in, directly and indirectly, a few dozen header files; the `#pragma hdrstop` is inserted to get better sharing with fewer PCH files. The PCH file produced for `comnfile.h` can be a bit over a megabyte in size. Another idiom, used by the source files involved in declaration processing, is this:

```
#include "comnfile.h"
#include "decl_hdrs.h"
#pragma hdrstop
#include ...
```

`decl_hdrs.h` pulls in another dozen header files, and a second, somewhat larger, PCH file is created. In all, the source files of a particular program can share just a few precompiled header files. If disk space were at a premium, you could decide to make `comnfile.h` pull in *all* the header files used -- then, a single PCH file could be used in building the program.

Different environments and different projects will have different needs, but in general, users should be aware that making the best use of the precompiled header support will require some experimentation and probably some minor changes to source code.

Chapter 3. Assembly Language

This chapter describes the most important aspects of the TASKING assembly language for TriCore. For a complete overview of the architecture you are using, refer to the target's core Architecture Manual.

3.1. Assembly Syntax

An assembly program consists of statements. A statement may optionally be followed by a comment. Any source statement can be extended to more lines by including the line continuation character (\) as the last character on the line. The length of a source statement (first line and continuation lines) is only limited by the amount of available memory.

Mnemonics, directives and other keywords are case insensitive. Labels, symbols, directive arguments, and literal strings are case sensitive.

The syntax of an assembly statement is:

```
[label[:]] [instruction | directive | macro_call] [;comment]
```

label	A label is a special symbol which is assigned the value and type of the current program location counter. A label can consist of letters, digits and underscore characters (_). The first character cannot be a digit. The label can also be a *number*. A label which is prefixed by whitespace (spaces or tabs) has to be followed by a colon (:). The size of an identifier is only limited by the amount of available memory.
	number is a number ranging from 1 to 255. This type of label is called a *numeric label* or *local label*. To refer to a numeric label, you must put an **n** (next) or **p** (previous) immediately after the label. This is required because the same label number may be used repeatedly.
	Examples:

```
   LAB1:   ; This label is followed by a colon and
           ; can be prefixed by whitespace
LAB1       ; This label has to start at the beginning
           ; of a line
1: j 1p    ; This is an endless loop
           ; using numeric labels
```

instruction	An instruction consists of a mnemonic and zero, one or more operands. It must not start in the first column.
	Operands are described in Section 3.3, *Operands of an Assembly Instruction*. The instructions are described in the target's core Architecture Manual.
directive	With directives you can control the assembler from within the assembly source. Except for preprocessing directives, these must not start in the first column. Directives are described in Section 3.9, *Assembler Directives and Controls*.

macro_call	A call to a previously defined macro. It must not start in the first column. See Section 3.10, *Macro Operations*.
comment	Comment, preceded by a ; (semicolon).

You can use empty lines or lines with only comments.

Apart from the assembly statements as described above, you can put a so-called 'control line' in your assembly source file. These lines start with a **$** in the first column and alter the default behavior of the assembler.

$`control`

For more information on controls see Section 3.9, *Assembler Directives and Controls*.

3.2. Assembler Significant Characters

You can use all ASCII characters in the assembly source both in strings and in comments. Also the extended characters from the ISO 8859-1 (Latin-1) set are allowed.

Some characters have a special meaning to the assembler. Special characters associated with expression evaluation are described in Section 3.6.3, *Expression Operators*. Other special assembler characters are:

Character	Description
;	Start of a comment
\	Line continuation character or macro operator: argument concatenation
?	Macro operator: return decimal value of a symbol
%	Macro operator: return hex value of a symbol
^	Macro operator: override local label
"	Macro string delimiter or quoted string `.DEFINE` expansion character
'	String constants delimiter
@	Start of a built-in assembly function
*	Location counter substitution
#	Constant number
++	String concatenation operator
[ ]	Substring delimiter

3.3. Operands of an Assembly Instruction

In an instruction, the mnemonic is followed by zero, one or more operands. An operand has one of the following types:

Operand	Description
symbol	A symbolic name as described in Section 3.4, *Symbol Names*. Symbols can also occur in expressions.
register	Any valid register as listed in Section 3.5, *Registers*.
expression	Any valid expression as described in Section 3.6, *Assembly Expressions*.
address	A combination of *expression*, *register* and *symbol*.

Addressing modes

The TriCore assembly language has several addressing modes. These are described in detail in the target's core Architecture Manual.

3.4. Symbol Names

User-defined symbols

A user-defined *symbol* can consist of letters, digits and underscore characters (_). The first character cannot be a digit. The size of an identifier is only limited by the amount of available memory. The case of these characters is significant. You can define a symbol by means of a label declaration or an equate or set directive.

Predefined preprocessor symbols

These symbols start and end with two underscore characters, __*symbol*__, and you can use them in your assembly source to create conditional assembly. See Section 3.4.1, *Predefined Preprocessor Symbols*.

Labels

Symbols used for memory locations are referred to as labels. It is allowed to use reserved symbols as labels as long as the label is followed by a colon.

Reserved symbols

Symbol names and other identifiers beginning with a period (.) are reserved for the system (for example for directives or section names). Instructions and registers are also reserved. The case of these built-in symbols is insignificant.

Examples

Valid symbol names:

```
loop_1
ENTRY
a_B_c
_aBC
```

Invalid symbol names:

```
1_loop      ; starts with a number
d15         ; reserved register name
.DEFINE     ; reserved directive name
```

3.4.1. Predefined Preprocessor Symbols

The TASKING assembler knows the predefined symbols as defined in the table below. The symbols are useful to create conditional assembly.

Symbol	Description
__ASTC__	Identifies the assembler. You can use this symbol to flag parts of the source which must be recognized by the **astc** assembler only. It expands to 1.
__BUILD__	Identifies the build number of the assembler, composed of decimal digits for the build number, three digits for the major branch number and three digits for the minor branch number. For example, if you use build 1.22.1 of the assembler, __BUILD__ expands to 1022001. If there is no branch number, the branch digits expand to zero. For example, build 127 results in 127000000.
__CORE_*core*__	A symbol is defined depending on the option option **--core=***core*. The *core* is converted to uppercase and '.' is removed. For example, if **--core=tc1.3.1** is specified, the symbol __CORE_TC131__ is defined. When no **--core** is supplied, the assembler defines __CORE_TC13__.
__CPU_*cpu*__	A symbol is defined depending on the control program option **--cpu=***cpu*. The *cpu* is converted to uppercase. For example, if **--cpu=tc1796b** is specified to the control program, the symbol __CPU_TC1796B__ is defined. When no **--cpu** is supplied, this symbol is not defined.
__CPU_TC*num*__	The corresponding symbol is defined if the CPU functional problem cpu-tc*num* is specified with the option **--silicon-bug**.
__FPU__	Expands to 0 if you used option **--no-fpu** (Disable FPU instructions), otherwise expands to 1.
__MMU__	Expands to 1 if you used option **--mmu-present** (allow use of MMU instructions), otherwise expands to 0.
__REVISION__	Expands to the revision number of the assembler. Digits are represented as they are; characters (for prototypes, alphas, betas) are represented by -1. Examples: v1.0r1 -> 1, v1.0rb -> -1
__TASKING__	Identifies the assembler as a TASKING assembler. Expands to 1 if a TASKING assembler is used.
__UM_KERNEL__	Expands to 1 if the TriCore runs in kernel/supervisor mode (option **--user-mode=kernel**).
__UM_USER_1__	Expands to 1 if the TriCore runs in User-1 mode (option **--user-mode=user-1**).

Symbol	Description
__VERSION__	Identifies the version number of the assembler. For example, if you use version 2.1r1 of the assembler, __VERSION__ expands to 2001 (dot and revision number are omitted, minor version number in 3 digits).

Example

```
.if @def('__ASTC__')
  ; this part is only for the astc assembler
...
.endif
```

3.5. Registers

The following register names, either uppercase or lowercase, should not be used for user-defined symbol names in an assembly language source file:

```
D0  .. D15   (data registers)
E0  .. E14   (data register pairs, only the even numbers)
A0  .. A15   (address registers)
```

3.5.1. Special Function Registers

It is easy to access Special Function Registers (SFRs) that relate to peripherals from assembly. The SFRs are defined in a special function register definition file (*.def) as symbol names for use by the assembler. The assembler can include the SFR definition file with the command line option **--include-file** (**-H**). SFRs are defined with .`EQU` directives.

For example (from `regtc1796b.def`):

```
PSW        .equ   0xFE04
```

Example use in assembly:

```
  mfcr    d0,#PSW
  andn    d0,d0,#0x7f             ; reset counter
  insert  d0,d0,#1,#7,#1          ; enable
  insert  d0,d0,#1,#8,#1          ; set GW bit
  mtcr    #PSW,d0
  isync
```

Without an SFR file the assembler only knows the general purpose registers D0-D15 and A0-A15.

3.6. Assembly Expressions

An expression is a combination of symbols, constants, operators, and parentheses which represent a value that is used as an operand of an assembler instruction (or directive).

Expressions can contain user-defined labels (and their associated integer or floating-point values), and any combination of integers, floating-point numbers, or ASCII literal strings.

Expressions follow the conventional rules of algebra and boolean arithmetic.

Expressions that can be evaluated at assembly time are called *absolute expressions.* Expressions where the result is unknown until all sections have been combined and located, are called *relocatable* or *relative expressions.*

When any operand of an expression is relocatable, the entire expression is relocatable. Relocatable expressions are emitted in the object file and evaluated by the linker. Relocatable expressions can only contain integral functions; floating-point functions and numbers are not supported by the ELF/DWARF object format.

The assembler evaluates expressions with 64-bit precision in two's complement.

The syntax of an *expression* can be any of the following:

- *numeric constant*

- *string*

- *symbol*

- *expression binary_operator expression*

- *unary_operator expression*

- **(***expression***)**

- *function call*

All types of expressions are explained in separate sections.

3.6.1. Numeric Constants

Numeric constants can be used in expressions. If there is no prefix, by default the assembler assumes the number is a decimal number. Prefixes can be used in either lowercase or uppercase.

Base	Description	Example
Binary	A **0b** or **0B** prefix followed by binary digits (0,1).	`0B1101` `0b11001010`

Hexadecimal	A **0x** or **0X** prefix followed by hexadecimal digits (0-9, A-F, a-f).	`0X12FF`
		`0x45`
		`0xfa10`

Decimal integer	Decimal digits (0-9).	`12` `1245`

Base	Description	Example

Decimal floating-point	Decimal digits (0-9), includes a decimal point, or an '**E**' or '**e**' followed by the exponent.	`6E10`
		`.6`
		`3.14`
		`2.7e10`

3.6.2. Strings

ASCII characters, enclosed in single (') or double (") quotes constitute an ASCII string. Strings between double quotes allow symbol substitution by a `.DEFINE` directive, whereas strings between single quotes are always literal strings. Both types of strings can contain escape characters.

Strings constants in expressions are evaluated to a number (each character is replaced by its ASCII value). Strings in expressions can have a size of up to 4 characters or less depending on the operand of an instruction or directive; any subsequent characters in the string are ignored. In this case the assembler issues a warning. An exception to this rule is when a string is used in a `.BYTE` assembler directive; in that case all characters result in a constant value of the specified size. Null strings have a value of 0.

Square brackets (**[ ]**) delimit a substring operation in the form:

```
[string,offset,length]
```

offset is the start position within string. *length* is the length of the desired substring. Both values may not exceed the size of *string*.

Examples

```
'ABCD'              ; (0x41424344)
'''79'              ; to enclose a quote double it
"A\"BC"             ; or to enclose a quote escape it
'AB'+1              ; (0x4143) string used in expression
''                  ; null string
.word 'abcdef'      ; (0x64636261) 'ef' are ignored
                    ; warning: string value truncated
'abc'++'de'         ; you can concatenate
                    ; two strings with the '++' operator.
                    ; This results in 'abcde'
['TASKING',0,4]     ; results in the substring 'TASK'
```

3.6.3. Expression Operators

The next table shows the assembler operators. They are ordered according to their precedence. Operators of the same precedence are evaluated left to right. Parenthetical expressions have the highest priority (innermost first).

Valid operands include numeric constants, literal ASCII strings and symbols.

Most assembler operators can be used with both integer and floating-point values. If one operand has an integer value and the other operand has a floating-point value, the integer is converted to a floating-point value before the operator is applied. The result is a floating-point value.

Type	Operator	Name	Description
	( )	parenthesis	Expressions enclosed by parenthesis are evaluated first.
Unary	+	plus	Returns the value of its operand.
	-	minus	Returns the negative of its operand.
	~	one's complement	Integer only. Returns the one's complement of its operand. It cannot be used with a floating-point operand.
	!	logical negate	Returns 1 if the operands' value is 0; otherwise 0. For example, if `buf` is 0 then `!buf` is 1. If `buf` has a value of 1000 then `!buf` is 0.
Arithmetic	*	multiplication	Yields the product of its operands.
	/	division	Yields the quotient of the division of the first operand by the second. For integer operands, the divide operation produces a truncated integer result.
	%	modulo	Integer only. This operator yields the remainder from the division of the first operand by the second.
	+	addition	Yields the sum of its operands.
	-	subtraction	Yields the difference of its operands.
Shift	<<	shift left	Integer only. Causes the left operand to be shifted to the left (and zero-filled) by the number of bits specified by the right operand.
	>>	shift right	Integer only. Causes the left operand to be shifted to the right by the number of bits specified by the right operand. The sign bit will be extended.
Relational	<	less than	Returns an integer 1 if the indicated condition is TRUE or an integer 0 if the indicated condition is FALSE.

For example, if D has a value of 3 and E has a value of 5, then the result of the expression `D<E` is 1, and the result of the expression `D>E` is 0.

Use tests for equality involving floating-point values with caution, since rounding errors could cause unexpected results. |
	<=	less than or equal	
	>	greater than	
	>=	greater than or equal	
	==	equal	
	!=	not equal	
Bit and Bitwise	:	bit position	Specify bit position (right operand) in a data register (left operand) for use in bit operations (instructions that have the .T data type modifier).

Type	Operator	Name	Description
	&	AND	Integer only. Yields the bitwise AND function of its operand.
	|	OR	Integer only. Yields the bitwise OR function of its operand.
	^	exclusive OR	Integer only. Yields the bitwise exclusive OR function of its operands.
Logical	&&	logical AND	Returns an integer 1 if both operands are non-zero; otherwise, it returns an integer 0.
	||	logical OR	Returns an integer 1 if either of the operands is non-zero; otherwise, it returns an integer 1

The relational operators and logical operators are intended primarily for use with the conditional assembly `.if` directive, but can be used in any expression.

3.7. Working with Sections

Sections are absolute or relocatable blocks of contiguous memory that can contain code or data. Some sections contain code or data that your program declared and uses directly, while other sections are created by the compiler or linker and contain debug information or code or data to initialize your application. These sections can be named in such a way that different modules can implement different parts of these sections. These sections are located in memory by the linker (using the linker script language, LSL) so that concerns about memory placement are postponed until after the assembly process.

All instructions and directives which generate data or code must be within an active section. The assembler emits a warning if code or data starts without a section definition and activation. The compiler automatically generates sections. If you program in assembly you have to define sections yourself.

For more information about locating sections see Section 7.8.9, *The Section Layout Definition: Locating Sections*.

Section definition

Sections are defined with the `.SDECL` directive and have a name. A section may have attributes to instruct the linker to place it on a predefined starting address, or that it may be overlaid with another section.

```
.SDECL "name",  type  [, attribute ]...  [AT address]
```

See the description of the `.SDECL directive` for a complete description of all possible attributes.

Section activation

Sections are defined once and are activated with the `.SECT` directive.

```
.SECT "name"
```

The linker will check between different modules and emits an error message if the section attributes do not match. The linker will also concatenate all matching section definitions into one section. So, all "code" sections generated by the compiler will be linked into one big "code" chunk which will be located in one piece. A .SECT directive referring to an earlier defined section is called a *continuation*. Only the name can be specified.

Examples

```
.SDECL  ".text.hello.main",CODE
.SECT   ".text.hello.main"
```

Defines and activates a relocatable section in CODE memory. Other parts of this section, with the same name, may be defined in the same module or any other module. Other modules should use the same .SDECL statement. When necessary, it is possible to give the section an absolute starting address.

```
.SDECL  ".CONST", CODE AT 0x1000
.SECT   ".CONST"
```

Defines and activates an absolute section named .CONST starting at address 0x1000.

```
.SDECL  ".fardata", DATA, CLEAR
.SECT   ".fardata"
```

Defines a relocatable named section in DATA memory. The CLEAR attribute instructs the linker to clear the memory located to this section. When this section is used in another module it must be defined identically. Continuations of this section in the same module are as follows:

```
.SECT   ".fardata"
```

3.8. Built-in Assembly Functions

The TASKING assembler has several built-in functions to support data conversion, string comparison, and math computations. You can use functions as terms in any expression.

Syntax of an assembly function

@*function_name*([*argument*[,*argument*]...])

Functions start with the '@' character and have zero or more arguments, and are always followed by opening and closing parentheses. White space (a blank or tab) is not allowed between the function name and the opening parenthesis and between the (comma-separated) arguments.

The names of assembly functions are case insensitive.

Overview of mathematical functions

Try to avoid usage of assembler functions that work with float values. The assembler uses IEEE floating-point routines of the host on which the assembler runs to calculate some fixed floating-point values. Because of the fact that there are differences between hosts (Windows, Linux and Solaris) with respect to the number of bits used and rounding mechanism (although all claim to be IEEE compliant) it is possible that some internal assembler functions return a slightly different value depending on the input. The difference is usually at position 16 behind the comma.

Function	Description
`@ABS(`*expr*`)`	Absolute value
`@ACS(`*expr*`)`	Arc cosine
`@ASN(`*expr*`)`	Arc sine
`@AT2(`*expr1*`,`*expr2*`)`	Arc tangent of *expr1* / *expr2*
`@ATN(`*expr*`)`	Arc tangent
`@CEL(`*expr*`)`	Ceiling function
`@COH(`*expr*`)`	Hyperbolic cosine
`@COS(`*expr*`)`	Cosine
`@FLR(`*expr*`)`	Floor function
`@L10(`*expr*`)`	Log base 10
`@LOG(`*expr*`)`	Natural logarithm
`@MAX(`*expr1*`[,...,`*exprN*`])`	Maximum value
`@MIN(`*expr1*`[,...,`*exprN*`])`	Minimum value
`@POW(`*expr1*`,`*expr2*`)`	Raise to a power
`@RND()`	Random value
`@SGN(`*expr*`)`	Returns the sign of an expression as -1, 0 or 1
`@SIN(`*expr*`)`	Sine
`@SNH(`*expr*`)`	Hyperbolic sine
`@SQT(`*expr*`)`	Square root
`@TAN(`*expr*`)`	Tangent
`@TNH(`*expr*`)`	Hyperbolic tangent
`@XPN(`*expr*`)`	Exponential function (raise e to a power)

Overview of conversion functions

Function	Description
`@CVF(`*expr*`)`	Convert integer to floating-point
`@CVI(`*expr*`)`	Convert floating-point to integer

Function	Description
`@FLD(`*base*`,`*value*`,`*width*`[,`*start*`])`	Shift and mask operation
`@FRACT(`*expr*`)`	Convert floating-point to 32-bit fractional
`@SFRACT(`*expr*`)`	Convert floating-point to 16-bit fractional
`@LNG(`*expr1*`,`*expr2*`)`	Concatenate to double word
`@LUN(`*expr*`)`	Convert long fractional to floating-point
`@RVB(`*expr*`[,`*exprN*`])`	Reverse order of bits in field
`@UNF(`*expr*`)`	Convert fractional to floating-point

Overview of string functions

Function	Description
`@CAT(`*str1*`,`*str2*`)`	Concatenate *str1* and *str2*
`@LEN(`*string*`)`	Length of string
`@POS(`*str1*`,`*str2*`[,`*start*`])`	Position of *str2* in *str1*
`@SCP(`*str1*`,`*str2*`)`	Compare *str1* with *str2*
`@SUB(`*str*`,`*expr1*`,`*expr2*`)`	Return substring

Overview of macro functions

Function	Description
`@ARG('`*symbol*`' | `*expr*`)`	Test if macro argument is present
`@CNT()`	Return number of macro arguments
`@MAC(`*symbol*`)`	Test if macro is defined
`@MXP()`	Test if macro expansion is active

Overview of address calculation functions

Function	Description
`@HI(`*expr*`)`	Returns upper 16 bits of expression value
`@HIS(`*expr*`)`	Returns upper 16 bits of expression value, adjusted for signed addition
`@LO(`*expr*`)`	Returns lower 16 bits of expression value
`@LOS(`*expr*`)`	Returns lower 16 bits of expression value, adjusted for signed addition
`@LSB(`*expr*`)`	Least significant byte of the expression
`@MSB(`*expr*`)`	Most significant byte of the expression

Overview of assembler mode functions

Function	Description
`@ASTC()`	Returns the name of the assembler executable
`@CPU('`*cpu*`')`	Test if CPU type is selected
`@DEF('`*symbol*`'` | *symbol*`)`	Returns 1 if symbol has been defined
`@EXP(`*expr*`)`	Expression check
`@INT(`*expr*`)`	Integer check
`@LST()`	LIST control flag value

Detailed Description of Built-in Assembly Functions

@ABS(*expression*)

Returns the absolute value of the expression as an integer value.

Example:

```
AVAL  .SET  @ABS(-2.1)  ; AVAL = 2
```

@ACS(*expression*)

Returns the arc cosine of *expression* as a floating-point value in the range zero to pi. The result of *expression* must be between -1 and 1.

Example:

```
ACOS  .SET  @ACS(-1.0)  ;ACOS = 3.1415926535897931
```

@ARG('*symbol*' | *expression*)

Returns integer 1 if the macro argument represented by *symbol* or *expression* is present, 0 otherwise.

You can specify the argument with a *symbol* name (the name of a macro argument enclosed in single quotes) or with *expression* (the ordinal number of the argument in the macro formal argument list). If you use this function when macro expansion is not active, the assembler issues a warning.

Example:

```
.IF @ARG('TWIDDLE') ;is argument twiddle present?
.IF @ARG(1)         ;is first argument present?
```

@ASN(*expression*)

Returns the arc sine of *expression* as a floating-point value in the range -pi/2 to pi/2. The result of *expression* must be between -1 and 1.

Example:

```
ARCSINE  .SET  @ASN(-1.0)        ;ARCSINE = -1.570796
```

@ASTC()

Returns the name of the assembler executable. This is 'astc' for the TriCore assembler.

Example:

```
ANAME:   .byte  @ASTC()          ;ANAME = 'astc'
```

@AT2(*expression1*,*expression2*)

Returns the arc tangent of *expression1*/*expression2* as a floating-point value in the range -pi to pi. *expression1* and *expression2* must be separated by a comma.

Example:

```
ATAN2  .EQU  @AT2(-1.0,1.0)    ;ATAN2 = -0.7853982
```

@ATN(*expression*)

Returns the arc tangent of *expression* as a floating-point value in the range -pi/2 to pi/2.

Example:

```
ATAN  .SET  @ATN(1.0)    ;ATAN = 0.78539816339744828
```

@CAT(*string1*,*string2*)

Concatenates the two strings into one string. The two strings must be enclosed in single or double quotes.

Example:

```
.DEFINE  ID  "@CAT('TASK','ING')"   ;ID = 'TASKING'
```

@CEL(*expression*)

Returns a floating-point value which represents the smallest integer greater than or equal to *expression*.

Example:

```
CEIL  .SET  @CEL(-1.05)          ;CEIL = -1.0
```

@CNT()

Returns the number of macro arguments of the current macro expansion as an integer. If you use this function when macro expansion is not active, the assembler issues a warning.

Example:

```
ARGCOUNT .SET @CNT() ; reserve argument count
```

@COH(*expression*)

Returns the hyperbolic cosine of *expression* as a floating-point value.

Example:

```
HYCOS  .EQU  @COH(VAL)          ;compute hyperbolic cosine
```

@COS(*expression*)

Returns the cosine of *expression* as a floating-point value.

Example:

```
.WORD  -@COS(@CVF(COUNT)*FREQ) ;compute cosine value
```

@CPU(*string*)

Returns integer 1 if *string* corresponds to the selected CPU type; 0 otherwise. See also control program option **--cpu** (Select CPU).

Example:

```
.IF @CPU("tc1.3.1")            ;TriCore 1.3.1 specific part
```

@CVF(*expression*)

Converts the result of *expression* to a floating-point value.

Example:

```
FLOAT  .SET  @CVF(5)           ;FLOAT = 5.0
```

@CVI(*expression*)

Converts the result of *expression* to an integer value. This function should be used with caution since the conversions can be inexact (e.g., floating-point values are truncated).

Example:

```
INT  .SET  @CVI(-1.05)         ;INT = -1
```

@DEF('*symbol*' | *symbol*)

Returns 1 if *symbol* has been defined, 0 otherwise. *symbol* can be any symbol or label not associated with a .MACRO or .SDECL directive. If *symbol* is quoted, it is looked up as a .DEFINE symbol; if it is not quoted, it is looked up as an ordinary symbol or label.

Example:

```
.IF @DEF('ANGLE')          ;is symbol ANGLE defined?
.IF @DEF(ANGLE)            ;does label ANGLE exist?
```

@EXP(*expression*)

Returns 0 if the evaluation of *expression* would normally result in an error. Returns 1 if the expression can be evaluated correctly. With the `@EXP` function, you prevent the assembler from generating an error if the expression contains an error. No test is made by the assembler for warnings. The expression may be relative or absolute.

Example:

```
.IF  !@EXP(3/0)        ;Do the IF on error
                       ;assembler generates no error

.IF  !(3/0)            ;assembler generates an error
```

@FLD(*base*,*value*,*width*[,*start*])

Shift and mask *value* into *base* for *width* bits beginning at bit *start*. If *start* is omitted, zero (least significant bit) is assumed. All arguments must be positive integers and none may be greater than the target word size. Returns the shifted and masked value.

Example:

```
VAR1 .EQU @FLD(0,1,1)   ;turn bit 0 on, VAR1=1
VAR2 .EQU @FLD(0,3,1)   ;turn bit 0 on, VAR2=1
VAR3 .EQU @FLD(0,3,2)   ;turn bits 0 and 1 on, VAR3=3
VAR4 .EQU @FLD(0,3,2,1) ;turn bits 1 and 2 on, VAR4=6
VAR5 .EQU @FLD(0,1,1,7) ;turn eighth bit on, VAR5=0x80
```

@FLR(*expression*)

Returns a floating-point value which represents the largest integer less than or equal to *expression*.

Example:

```
FLOOR    .SET  @FLR(2.5)        ;FLOOR = 2.0
```

@FRACT(*expression*)

Returns the 32-bit fractional representation of the floating-point *expression*. The expression must be in the range [-1,+1>.

Example:

```
.WORD  @FRACT(0.1), @FRACT(1.0)
```

@HI(*expression*)

Returns the upper 16 bits of a value. @HI(expression) is equivalent to ((expression>>16) & 0xffff).

Example:

```
mov.u   d2,#@LO(COUNT)
addih   d2,d2,#@HI(COUNT)      ;upper 16 bits of COUNT
```

@HIS(*expression*)

Returns the upper 16 bits of a value, adjusted for a signed addition. @HIS(expression) is equivalent to (((expression+0x800)>>16) & 0xffff).

Example:

```
movh.a  a3,#@HIS(label)
lea     a3,[a3]@LOS(label)
```

@INT(*expression*)

Returns integer 1 if *expression* has an integer result; otherwise, it returns a 0. The expression may be relative or absolute.

Example:

```
.IF  @INT(TERM)     ;Test if result is an integer
```

@L10(*expression*)

Returns the base 10 logarithm of *expression* as a floating-point value. *expression* must be greater than zero.

Example:

```
LOG    .EQU  @L10(100.0)      ;LOG = 2
```

@LEN(*string*)

Returns the length of *string* as an integer.

Example:

```
SLEN   .SET  @LEN('string')   ;SLEN = 6
```

@LNG(*expression1*,*expression2*)

Concatenates the 16-bit *expression1* and *expression2* into a 32-bit word value such that *expression1* is the high half and *expression2* is the low half.

Example:

```
LWORD  .WORD  @LNG(HI,LO)        ;build long word
```

@LO(*expression*)

Returns the lower 16 bits of a value. `@LO(expression)` is equivalent to `(expression & 0xffff)`.

Example:

```
mov.u   d2,#@LO(COUNT)         ;lower 16 bits of COUNT
addih   d2,d2,#@HI(COUNT)
```

@LOG(*expression*)

Returns the natural logarithm of *expression* as a floating-point value. *expression* must be greater than zero.

Example:

```
LOG    .EQU  @LOG(100.0)       ;LOG = 4.605170
```

@LOS(*expression*)

Returns the lower 16 bits of a value, adjusted for a signed addition. `@LOS(expression)` is equivalent to `(((expression+0x8000) & 0xffff) - 0x8000)`.

Example:

```
movh.a   a3,#@HIS(label)
lea      a3,[a3]@LOS(label)
```

@LSB(*expression*)

Returns the least significant byte of the result of the *expression*. The result of the expression is calculated as 16 bit.

Example:

```
VAR1   .SET @LSB(0x34)        ;VAR1 = 0x34
VAR2   .SET @LSB(0x1234)      ;VAR2 = 0x34
VAR3   .SET @LSB(0x654321)    ;VAR3 = 0x21
```

@LST()

Returns the value of the `$LIST ON/OFF` control flag as an integer. Whenever a `$LIST ON` control is encountered in the assembler source, the flag is incremented; when a `$LIST OFF` control is encountered, the flag is decremented.

Example:

```
.DUP    @ABS(@LST())            ;list unconditionally
```

@LUN(*expression*)

Converts the 32-bit *expression* to a floating-point value. *expression* should represent a binary fraction.

Example:

```
DBLFRC1 .EQU  @LUN(0x40000000)  ;DBLFRC1 = 0.5
DBLFRC2 .EQU  @LUN(3928472)     ;DBLFRC2 = 0.007354736
DBLFRC3 .EQU  @LUN(0xE0000000)  ;DBLFRC3 = -0.75
```

@MAC(*symbol*)

Returns integer 1 if *symbol* has been defined as a macro name, 0 otherwise.

Example:

```
.IF     @MAC(DOMUL)             ;does macro DOMUL exist?
```

@MAX(*expression1*[,*expressionN*],...)

Returns the maximum value of *expression1*, ..., *expressionN* as a floating-point value.

Example:

```
MAX:    .BYTE @MAX(1,-2.137,3.5)  ;MAX = 3.5
```

@MIN(*expression1*[,*expressionN*],...)

Returns the minimum value of *expression1*, ..., *expressionN* as a floating-point value.

Example:

```
MIN:    .BYTE @MIN(1,-2.137,3.5)  ;MIN = -2.137
```

@MSB(*expression*)

Returns the most significant byte of the result of the *expression*. The result of the expression is calculated as 16 bit.

Example:

```
VAR1    .SET @MSB(0x34)         ;VAR1 = 0x00
VAR2    .SET @MSB(0x1234)       ;VAR2 = 0x12
VAR3    .SET @MSB(0x654321)     ;VAR3 = 0x43
```

@MXP()

Returns integer 1 if the assembler is expanding a macro, 0 otherwise.

Example:

```
.IF    @MXP()                    ;macro expansion active?
```

@POS(*string1,string2*[*,start*])

Returns the position of *string2* in *string1* as an integer. If *string2* does not occur in *string1*, the last string position + 1 is returned.

With *start* you can specify the starting position of the search. If you do not specify start, the search is started from the beginning of *string1*. Note that the first position in a string is position 0.

Example:

```
ID1  .EQU  @POS('TASKING','ASK')  ; ID1 = 1
ID2  .EQU  @POS('ABCDABCD','B',2) ; ID2 = 5
ID3  .EQU  @POS('TASKING','BUG')  ; ID3 = 7
```

@POW(*expression1,expression2*)

Returns *expression1* raised to the power *expression2* as a floating-point value. *expression1* and *expression2* must be separated by a comma.

Example:

```
BUF  .EQU  @CVI(@POW(2.0,3.0))     ;BUF = 8
```

@RND()

Returns a random value in the range 0.0 to 1.0.

Example:

```
SEED  .EQU   @RND()          ;save initial SEED value
```

@RVB(*expression1,expression2*)

Reverse the order of bits in *expression1* delimited by the number of bits in *expression2*. If *expression2* is omitted the field is bounded by the target word size. Both expressions must be 16-bit integer values.

Example:

```
VAR1 .SET @RVB(0x200)    ;reverse all bits, VAR1=0x40
VAR2 .SET @RVB(0xB02)    ;reverse all bits, VAR2=0x40D0
VAR3 .SET @RVB(0xB02,2)  ;reverse bits 0 and 1,
                         ;VAR3=0xB01
```

@SCP(*string1,string2*)

Returns integer 1 if the two strings compare, 0 otherwise. The two strings must be separated by a comma.

Example:

```
.IF @SCP(STR,'MAIN')  ; does STR equal 'MAIN'?
```

@SFRACT(*expression*)

This function returns the 16-bit fractional representation of the floating-point expression. The expression must be in the range [-1,+1>.

Example:

```
.WORD  @SFRACT(0.1), @SFRACT(1.0)
```

@SGN(*expression*)

Returns the sign of *expression* as an integer: -1 if the argument is negative, 0 if zero, 1 if positive. The expression may be relative or absolute.

Example:

```
VAR1  .SET @SGN(-1.2e-92)   ;VAR1 = -1
VAR2  .SET @SGN(0)          ;VAR2 =  0
VAR3  .SET @SGN(28.382)     ;VAR3 =  1
```

@SIN(*expression*)

Returns the sine of *expression* as a floating-point value.

Example:

```
.WORD @SIN(@CVF(COUNT)*FREQ)   ;compute sine value
```

@SNH(*expression*)

Returns the hyperbolic sine of *expression* as a floating-point value.

Example:

```
HSINE .EQU  @SNH(VAL)           ;hyperbolic sine
```

@SQT(*expression*)

Returns the square root of *expression* as a floating-point value. *expression* must be positive.

Example:

```
SQRT1  .EQU  @SQT(3.5)          ;SQRT1 = 1.870829
SQRT2  .EQU  @SQT(16)           ;SQRT2 = 4
```

@SUB(*string*,*expression1*,*expression2*)

Returns the substring from *string* as a string. *expression1* is the starting position within *string*, and *expression2* is the length of the desired string. The assembler issues an error if either *expression1* or *expression2* exceeds the length of string. Note that the first position in a string is position 0.

Example:

```
.DEFINE  ID  "@SUB('TASKING',3,4)"  ;ID = 'KING'
```

@TAN(*expression*)

Returns the tangent of *expression* as a floating-point value.

Example:

```
TANGENT .SET @TAN(1.0)           ;TANGENT = 1.5574077
```

@TNH(*expression*)

Returns the hyperbolic tangent of *expression* as a floating-point value.

Example:

```
HTAN  .SET  @TNH(1)              ;HTAN = 0.76159415595
```

@UNF(*expression*)

Converts *expression* to a floating-point value. *expression* should represent a 16-bit binary fraction.

Example:

```
FRC   .EQU  @UNF(0x4000)         ;FRC = 0.5
```

@XPN(*expression*)

Returns the exponential function (base e raised to the power of *expression*) as a floating-point value.

Example:

```
EXP   .EQU  @XPN(1.0)            ;EXP = 2.718282
```

3.9. Assembler Directives and Controls

An assembler directive is simply a message to the assembler. Assembler directives are not translated into machine instructions. There are three main groups of assembler directives.

• Assembler directives that tell the assembler how to go about translating instructions into machine code. This is the most typical form of assembly directives. Typically they tell the assembler where to put a program in memory, what space to allocate for variables, and allow you to initialize memory with data. When the assembly source is assembled, a location counter in the assembler keeps track of where the code and data is to go in memory.

  The following directives fall under this group:

  • Assembly control directives

  • Symbol definition and section directives

  • Data definition / Storage allocation directives

  • High Level Language (HLL) directives

• Directives that are interpreted by the macro preprocessor. These directives tell the macro preprocessor how to manipulate your assembly code before it is actually being assembled. You can use these directives to write macros and to write conditional source code. Parts of the code that do not match the condition, will not be assembled at all.

• Some directives act as assembler options and most of them indeed do have an equivalent assembler (command line) option. The advantage of using a directive is that with such a directive you can overrule the assembler option for a particular part of the code. Directives of this kind are called *controls*. A typical example is to tell the assembler with an option to generate a list file while with the controls $LIST ON and $LIST OFF you overrule this option for a part of the code that you do not want to appear in the list file. Controls always appear on a separate line and start with a '$' sign in the first column.

  The following controls are available:

  • Assembly listing controls

  • Miscellaneous controls

Each assembler directive or control has its own syntax. You can use assembler directives and controls in the assembly code as pseudo instructions.

Some assembler directives can be preceded with a label. If you do not precede an assembler directive with a label, you must use white space instead (spaces or tabs). The assembler recognizes both uppercase and lowercase for directives.

3.9.1. Assembler Directives

Overview of assembly control directives

Directive	Description
.COMMENT	Start comment lines. You cannot use this directive in .IF/.ELSE/.ENDIF constructs and .MACRO/.DUP definitions.
.END	Indicates the end of an assembly module
.FAIL	Programmer generated error message
.INCLUDE	Include file
.MESSAGE	Programmer generated message
.WARNING	Programmer generated warning message

Overview of symbol definition and section directives

Directive	Description
.ALIAS	Create an alias for a symbol
.EQU	Set permanent value to a symbol
.EXTERN	Import global section symbol
.GLOBAL	Declare global section symbol
.LOCAL	Declare local section symbol
.NAME	Specify name of original C source file
.ORG	Initialize memory space and location counters to create a nameless section
.SDECL	Declare a section with name, type and attributes
.SECT	Activate a declared section
.SET	Set temporary value to a symbol
.SIZE	Set size of symbol in the ELF symbol table
.TYPE	Set symbol type in the ELF symbol table
.WEAK	Mark a symbol as 'weak'

Overview of data definition / storage allocation directives

Directive	Description
.ACCUM	Define 64-bit constant of 18 + 46 bits format
.ALIGN	Align location counter
.ASCII, .ASCIIZ	Define ASCII string without / with ending NULL byte
.BYTE	Define byte
.DOUBLE	Define a 64-bit floating-point constant
.FLOAT	Define a 32-bit floating-point constant

Directive	Description
`.FRACT`	Define a 32-bit constant fraction
`.HALF`	Define half-word (16 bits)
`.SFRACT`	Define a 16-bit constant fraction
`.SPACE`	Define storage
`.WORD`	Define word (32 bits)

Overview of macro preprocessor directives

Directive	Description
`.DEFINE`	Define substitution string
`.DUP`, `.ENDM`	Duplicate sequence of source lines
`.DUPA`, `.ENDM`	Duplicate sequence with arguments
`.DUPC`, `.ENDM`	Duplicate sequence with characters
`.DUPF`, `.ENDM`	Duplicate sequence in loop
`.IF`, `.ELIF`, `.ELSE`	Conditional assembly directive
`.ENDIF`	End of conditional assembly directive
`.EXITM`	Exit macro
`.MACRO`, `.ENDM`	Define macro
`.PMACRO`	Undefine (purge) macro
`.UNDEF`	Undefine `.DEFINE` symbol

Overview of HLL directives

Directive	Description
`.CALLS`	Pass call tree information and/or stack usage information
`.MISRAC`	Pass MISRA C information

.ACCUM

Syntax

[*label*:]**.ACCUM** *expression*[**,***expression*]...

Description

With the .ACCUM directive the assembler allocates and initializes two words of memory (64 bits) for each argument. Use commas to separate multiple arguments.

An *expression* can be:

- a fractional fixed point expression (range [-2^{17}, 2^{17}>)

- NULL (indicated by two adjacent commas: ,,)

Multiple arguments are stored in successive address locations in sets of two words. If an argument is NULL its corresponding address location is filled with zeros.

If the evaluated expression is out of the range [-2^{17}, 2^{17}>, the assembler issues a warning and saturates the fractional value.

Example

ACC: .ACCUM 0.1,0.2,0.3

Related Information

.FRACT, .SFRACT (Define 32-bit/16-bit constant fraction)

.SPACE (Define Storage)

.ALIAS

Syntax

alias-name **.ALIAS** *symbol-name*

Description

With the .ALIAS directive you can create an alias of a symbol. The C compiler generates this directive when you use the #pragma alias.

Example

```
exit .ALIAS _Exit
```

Related information

Pragma alias

.ALIGN

Syntax

```
.ALIGN expression
```

Description

With the `.ALIGN` directive you instruct the assembler to align the location counter. By default the assembler aligns on one byte.

When the assembler encounters the `.ALIGN` directive, it advances the location counter to an address that is aligned as specified by *expression* and places the next instruction or directive on that address. The alignment is in minimal addressable units (MAUs). The assembler fills the 'gap' with NOP instructions for code sections or with zeros for data sections. If the location counter is already aligned on the specified alignment, it remains unchanged. The location of absolute sections will not be changed.

The *expression* must be a power of two: 2, 4, 8, 16, ... If you specify another value, the assembler changes the alignment to the next higher power of two and issues a warning.

The assembler aligns sections automatically to the largest alignment value occurring in that section.

A label is not allowed before this directive.

Example

```
.sdecl '.text.mod.csec',code
.sect  '.text.mod.csec'
.ALIGN 16    ; the assembler aligns
instruction  ; this instruction at 16 MAUs and
             ; fills the 'gap' with NOP instructions.

.sdecl '.text.mod.csec2',code
.sect  '.text.mod.csec2'
.ALIGN 12    ; WRONG: not a power of two, the
instruction  ; assembler aligns this instruction at
             ; 16 MAUs and issues a warning.
```

.ASCII, .ASCIIZ

Syntax

[*label***:**] **.ASCII** *string*[**,***string*]...

[*label***:**] **.ASCIIZ** *string*[**,***string*]...

Description

With the .ASCII or .ASCIIZ directive the assembler allocates and initializes memory for each *string* argument.

The .ASCII directive does not add a NULL byte to the end of the string. The .ASCIIZ directive does add a NULL byte to the end of the string. The "z" in .ASCIIZ stands for "zero". Use commas to separate multiple strings.

Example

```
STRING:  .ASCII  "Hello world"
STRINGZ: .ASCIIZ "Hello world"
```

Note that with the .BYTE directive you can obtain exactly the same effect:

```
STRING:  .BYTE  "Hello world"    ; without a NULL byte
STRINGZ: .BYTE  "Hello world",0  ; with a NULL byte
```

Related Information

.BYTE (Define a constant byte)

.SPACE (Define Storage)

.BYTE

Syntax

```
[label:] .BYTE argument[,argument]...
```

Description

With the `.BYTE` directive the assembler allocates and initializes a byte of memory for each argument.

An *argument* can be:

- a single or multiple character string constant

- an integer expression

- NULL (indicated by two adjacent commas: ,,)

Multiple arguments are stored in successive byte locations. If an argument is NULL its corresponding byte location is filled with zeros.

If you specify *label*, it gets the value of the location counter at the start of the directive processing.

Integer arguments are stored as is, but must be byte values (within the range 0-255); floating-point numbers are not allowed. If the evaluated expression is out of the range [-256, +255] the assembler issues an error. For negative values within that range, the assembler adds 256 to the specified value (for example, -254 is stored as 2).

String constants

Single-character strings are stored in a byte whose lower seven bits represent the ASCII value of the character, for example:

```
.BYTE 'R'        ; = 0x52
```

Multiple-character strings are stored in consecutive byte addresses, as shown below. The standard C language escape characters like '\n' are permitted.

```
.BYTE 'AB',,'C'  ; = 0x41420043 (second argument is empty)
```

Example

```
TABLE  .BYTE  'two',0,'strings',0
CHARS  .BYTE  'A','B','C','D'
```

Related Information

.ASCII, .ASCIIZ (Define ASCII string without/with ending NULL)

.WORD, .HALF (Define a word / halfword)

.SPACE (Define Storage)

.CALLS

Syntax

.CALLS *'caller','callee'*

or

.CALLS *'caller','',stack_usage*[,...]

Description

The first syntax creates a call graph reference between *caller* and *callee*. The linker needs this information to build a call graph. *caller* and *callee* are names of functions.

The second syntax specifies stack information. When *callee* is an empty name, this means we define the stack usage of the function itself. The value specified is the stack usage in bytes at the time of the call including the return address. A function can use multiple stacks.

This information is used by the linker to compute the used stack within the application. The information is found in the generated linker map file within the Memory Usage.

This directive is generated by the C compiler. Normally you will not use it in hand-coded assembly.

A label is not allowed before this directive.

Example

```
.CALLS 'main','nfunc'
```

Indicates that the function `main` calls the function `nfunc`.

```
.CALLS 'main','',8
```

The function `main` uses 8 bytes on the stack.

.COMMENT

Syntax

```
.COMMENT  delimiter
.
.
delimiter
```

Description

With the `.COMMENT` directive you can define one or more lines as comments. The first non-blank character after the `.COMMENT` directive is the comment delimiter. The two delimiters are used to define the comment text. The line containing the second comment delimiter will be considered the last line of the comment. The comment text can include any printable characters and the comment text will be produced in the source listing as it appears in the source file.

A label is not allowed before this directive.

Example

```
.COMMENT  + This is a one line comment +
.COMMENT  * This is a multiple line
            comment. Any number of lines
            can be placed between the two
            delimiters.
          *
```

.DEFINE

Syntax

```
.DEFINE symbol  string
```

Description

With the `.DEFINE` directive you define a substitution string that you can use on all following source lines. The assembler searches all succeeding lines for an occurrence of *symbol*, and replaces it with *string*. If the *symbol* occurs in a double quoted string it is also replaced. Strings between single quotes are not expanded.

This directive is useful for providing better documentation in the source program. A *symbol* can consist of letters, digits and underscore characters (_), and the first character cannot be a digit.

Macros represent a special case. `.DEFINE` directive translations will be applied to the macro definition as it is encountered. When the macro is expanded, any active `.DEFINE` directive translations will again be applied.

The assembler issues a warning if you redefine an existing symbol.

A label is not allowed before this directive.

Example

Suppose you defined the symbol LEN with the substitution string "32":

```
.DEFINE LEN "32"
```

Then you can use the symbol LEN for example as follows:

```
.SPACE LEN
.MESSAGE "The length is: LEN"
```

The assembler preprocessor replaces LEN with "32" and assembles the following lines:

```
.SPACE 32
.MESSAGE "The length is: 32"
```

Related Information

.UNDEF (Undefine a .DEFINE symbol)

.MACRO, .ENDM (Define a macro)

.DUP, .ENDM

Syntax

```
[label:] .DUP expression
    ....
    .ENDM
```

Description

With the .DUP/.ENDM directive you can duplicate a sequence of assembly source lines. With *expression* you specify the number of duplications. If the *expression* evaluates to a number less than or equal to 0, the sequence of lines will not be included in the assembler output. The *expression* result must be an absolute integer and cannot contain any forward references (symbols that have not already been defined). The .DUP directive may be nested to any level.

If you specify *label*, it gets the value of the location counter at the start of the directive processing.

Example

In this example the loop is repeated three times. Effectively, the preprocessor repeats the source lines (.BYTE 10) three times, then the assembler assembles the result:

```
.DUP 3
.BYTE 10  ; assembly source lines
.ENDM
```

Related Information

.DUPA, .ENDM (Duplicate sequence with arguments)

.DUPC, .ENDM (Duplicate sequence with characters)

.DUPF, .ENDM (Duplicate sequence in loop)

.MACRO, .ENDM (Define a macro)

.DUPA, .ENDM

Syntax

```
[label:]  .DUPA  formal_arg,argument[,argument]...
    ....
    .ENDM
```

Description

With the `.DUPA`/`.ENDM` directive you can repeat a block of source statements for each *argument*. For each repetition, every occurrence of the *formal_arg* parameter within the block is replaced with each succeeding *argument* string. If an argument includes an embedded blank or other assembler-significant character, it must be enclosed with single quotes.

If you specify *label*, it gets the value of the location counter at the start of the directive processing.

Example

Consider the following source input statements,

```
.DUPA  VALUE,12,,32,34
.BYTE  VALUE
.ENDM
```

This is expanded as follows:

```
.BYTE  12
.BYTE  VALUE  ; results in a warning
.BYTE  32
.BYTE  34
```

The second statement results in a warning of the assembler that the local symbol `VALUE` is not defined in this module and is made external.

Related Information

`.DUP, .ENDM` (Duplicate sequence of source lines)

`.DUPC, .ENDM` (Duplicate sequence with characters)

`.DUPF, .ENDM` (Duplicate sequence in loop)

`.MACRO, .ENDM` (Define a macro)

.DUPC, .ENDM

Syntax

```
[label:] .DUPC formal_arg,string
    ....
    .ENDM
```

Description

With the `.DUPC`/`.ENDM` directive you can repeat a block of source statements for each character within *string*. For each character in the string, the *formal_arg* parameter within the block is replaced with that character. If the string is empty, then the block is skipped.

If you specify *label*, it gets the value of the location counter at the start of the directive processing.

Example

Consider the following source input statements,

```
.DUPC   VALUE,'123'
.BYTE   VALUE
.ENDM
```

This is expanded as follows:

```
.BYTE   1
.BYTE   2
.BYTE   3
```

Related Information

`.DUP, .ENDM` (Duplicate sequence of source lines)

`.DUPA, .ENDM` (Duplicate sequence with arguments)

`.DUPF, .ENDM` (Duplicate sequence in loop)

`.MACRO, .ENDM` (Define a macro)

.DUPF, .ENDM

Syntax

```
[label:]  .DUPF formal_arg,[start],end[,increment]
    ....
    .ENDM
```

Description

With the `.DUPF`/`.ENDM` directive you can repeat a block of source statements (*end* - *start*) + 1 / *increment* times. *start* is the starting value for the loop index; *end* represents the final value. *increment* is the increment for the loop index; it defaults to 1 if omitted (as does the *start* value). The *formal_arg* parameter holds the loop index value and may be used within the body of instructions.

If you specify *label*, it gets the value of the location counter at the start of the directive processing.

Example

Consider the following source input statements,

```
.DUPF    NUM,0,7
MOV  D\NUM,#0
.ENDM
```

This is expanded as follows:

```
MOV  D0,#0
MOV  D1,#0
MOV  D2,#0
MOV  D3,#0
MOV  D4,#0
MOV  D5,#0
MOV  D6,#0
MOV  D7,#0
```

Related Information

`.DUP, .ENDM` (Duplicate sequence of source lines)

`.DUPA, .ENDM` (Duplicate sequence with arguments)

`.DUPC, .ENDM` (Duplicate sequence with characters)

`.MACRO, .ENDM` (Define a macro)

.END

Syntax

.END

Description

With the optional .END directive you tell the assembler that the end of the module is reached. If the assembler finds assembly source lines beyond the .END directive, it ignores those lines and issues a warning.

You cannot use the .END directive in a macro expansion.

The assembler does not allow a label with this directive.

When you use assembler option **--require-end**, the .END directive is mandatory.

Example

```
     ; source lines
  .END                 ; End of assembly module
```

Related Information

-

.EQU

Syntax

symbol **.EQU** *expression*

Description

With the `.EQU` directive you assign the value of *expression* to *symbol* permanently. The expression can be relocatable or absolute and forward references are allowed. Once defined, you cannot redefine the symbol. With the `.GLOBAL` directive you can declare the symbol global.

Example

To assign the value 0x400 permanently to the symbol MYSYMBOL:

```
MYSYMBOL .EQU  0x4000
```

You cannot redefine the symbol MYSYMBOL after this.

Related Information

`.SET` (Set temporary value to a symbol)

.EXITM

Syntax

```
.EXITM
```

Description

With the `.EXITM` directive the assembler will immediately terminate a macro expansion. It is useful when you use it with the conditional assembly directive `.IF` to terminate macro expansion when, for example, error conditions are detected.

A label is not allowed before this directive.

Example

```
CALC  .MACRO  XVAL,YVAL
      .IF     XVAL<0
      .FAIL   'Macro parameter value out of range'
      .EXITM  ;Exit macro
      .ENDIF
        .
        .
        .
      .ENDM
```

Related Information

`.DUP, .ENDM` (Duplicate sequence of source lines)

`.DUPA, .ENDM` (Duplicate sequence with arguments)

`.DUPC, .ENDM` (Duplicate sequence with characters)

`.DUPF, .ENDM` (Duplicate sequence in loop)

`.MACRO, .ENDM` (Define a macro)

.EXTERN

Syntax

```
.EXTERN symbol[,symbol]...
```

Description

With the .EXTERN directive you define an *external* symbol. It means that the specified symbol is referenced in the current module, but is not defined within the current module. This symbol must either have been defined outside of any module or declared as globally accessible within another module with the .GLOBAL directive.

If you do not use the .EXTERN directive and the symbol is not defined within the current module, the assembler issues a warning and inserts the .EXTERN directive.

A label is not allowed with this directive.

Example

```
.EXTERN AA,CC,DD       ;defined elsewhere
.sdecl ".text.code", code
.sect  ".text.code"
.
.
MOV D0, #AA    ; AA is used here
.
```

Related Information

.GLOBAL (Declare global section symbol)

.LOCAL (Declare local section symbol)

.FAIL

Syntax

```
.FAIL {str|exp}[,{str|exp}]...
```

Description

With the .FAIL directive you tell the assembler to print an error message to stderr during the assembling process.

An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be specified to describe the nature of the generated error. If you use expressions, the assembler outputs the result. The assembler outputs a space between each argument.

The total error count will be incremented as with any other error. The .FAIL directive is for example useful in combination with conditional assembly for exceptional condition checking. The assembly process proceeds normally after the error has been printed.

With this directive the assembler exits with exit code 1 (an error).

A label is not allowed with this directive.

Example

```
.FAIL  'Parameter out of range'
```

This results in the error:

```
E143: ["filename" line] Parameter out of range
```

Related Information

.MESSAGE (Programmer generated message)

.WARNING (Programmer generated warning)

.FLOAT, .DOUBLE

Syntax

[*label*:].**FLOAT** *expression*[**,***expression*]...

[*label*:].**DOUBLE** *expression*[**,***expression*]...

Description

With the .FLOAT or .DOUBLE directive the assembler allocates and initializes a floating-point number (32 bits) or a double (64 bits) in memory for each argument.

An *expression* can be:

• a floating-point expression

• NULL (indicated by two adjacent commas: ,,)

You can represent a constant as a signed whole number with fraction or with the 'e' format as used in the C language. For example, 12.457 and +0.27E-13 are legal floating-point constants.

If the evaluated argument is too large to be represented in a single word / double-word, the assembler issues an error and truncates the value.

If you specify *label*, it gets the value of the location counter at the start of the directive processing.

Example

```
FLT:  .FLOAT   12.457,+0.27E-13
DBL:  .DOUBLE  12.457,+0.27E-13
```

Related Information

.SPACE (Define Storage)

.FRACT, .SFRACT

Syntax

[*label*:].**FRACT** *expression*[**,***expression*]...

[*label*:].**SFRACT** *expression*[**,***expression*]...

Description

With the `.FRACT` or `.SFRACT` directive the assembler allocates and initializes a 32-bit or 16-bit constant fraction in memory for each argument. Use commas to separate multiple arguments.

An *expression* can be:

- a fractional fixed point expression (range [-1, +1>)

- NULL (indicated by two adjacent commas: ,,)

Multiple arguments are stored in successive address locations in sets of two bytes. If an argument is NULL its corresponding address location is filled with zeros.

If the evaluated expression is out of the range [-1, +1>, the assembler issues a warning and saturates the fractional value.

Example

```
FRCT:    .FRACT   0.1,0.2,0.3
SFRCT:   .SFRACT  0.1,0.2,0.3
```

Related Information

`.ACCUM` (Define 64-bit constant fraction in 18+46 bits format)

`.SPACE` (Define Storage)

.GLOBAL

Syntax

```
.GLOBAL symbol[,symbol]...
```

Description

All symbols or labels defined in the current section or module are local to the module by default. You can change this default behavior with assembler option **--symbol-scope=global**.

With the .GLOBAL directive you declare one of more symbols as global. It means that the specified symbols are defined within the current section or module, and that those definitions should be accessible by all modules.

To access a symbol, defined with .GLOBAL, from another module, use the .EXTERN directive.

Only program labels and symbols defined with .EQU can be made global.

If the symbols that appear in the operand field are not used in the module, the assembler gives a warning.

The assembler does not allow a label with this directive.

Example

```
LOOPA .EQU 1          ; definition of symbol LOOPA
      .GLOBAL  LOOPA  ; LOOPA will be globally
                      ; accessible by other modules
```

Related Information

.EXTERN (Import global section symbol)

.LOCAL (Declare local section symbol)

.IF, .ELIF, .ELSE, .ENDIF

Syntax

```
.IF   expression
 .
 .
[.ELIF  expression]  ; the .ELIF directive is optional
 .
 .
[.ELSE]              ; the .ELSE directive is optional
 .
 .
.ENDIF
```

Description

With the `.IF`/`.ENDIF` directives you can create a part of conditional assembly code. The assembler assembles only the code that matches a specified condition.

The *expression* must evaluate to an absolute integer and cannot contain forward references. If *expression* evaluates to zero, the IF-condition is considered FALSE, any non-zero result of *expression* is considered as TRUE.

If the optional `.ELSE` and/or `.ELIF` directives are not present, then the source statements following the `.IF` directive and up to the next `.ENDIF` directive will be included as part of the source file being assembled only if the *expression* had a non-zero result.

If the *expression* has a value of zero, the source file will be assembled as if those statements between the `.IF` and the `.ENDIF` directives were never encountered.

If the `.ELSE` directive is present and expression has a nonzero result, then the statements between the `.IF` and `.ELSE` directives will be assembled, and the statement between the `.ELSE` and `.ENDIF` directives will be skipped. Alternatively, if expression has a value of zero, then the statements between the `.IF` and `.ELSE` directives will be skipped, and the statements between the `.ELSE` and `.ENDIF` directives will be assembled.

You can nest `.IF` directives to any level. The `.ELSE` and `.ELIF` directive always refer to the nearest previous `.IF` directive.

A label is not allowed with this directive.

Example

Suppose you have an assemble source file with specific code for a test version, for a demo version and for the final version. Within the assembly source you define this code conditionally as follows:

```
.IF    TEST
... ; code for the test version
.ELIF DEMO
... ; code for the demo version
.ELSE
```

```
  ... ; code for the final version
  .ENDIF
```

Before assembling the file you can set the values of the symbols TEST and DEMO in the assembly source before the .IF directive is reached. For example, to assemble the demo version:

```
TEST .SET 0
DEMO .SET 1
```

You can also define the symbols on the command line with the assembler option **--define** (**-D**):

```
astc --define=DEMO --define=TEST=0 test.asm
```

.INCLUDE

Syntax

.INCLUDE "*filename*" | <*filename*>

Description

With the .INCLUDE directive you include another file at the exact location where the .INCLUDE occurs. This happens before the resulting file is assembled. The .INCLUDE directive works similarly to the #include statement in C. The source from the include file is assembled as if it followed the point of the .INCLUDE directive. When the end of the included file is reached, assembly of the original file continues.

The string specifies the filename of the file to be included. The filename must be compatible with the operating system (forward/backward slashes) and can contain a directory specification.

If an absolute pathname is specified, the assembler searches for that file. If a relative path is specified or just a filename, the order in which the assembler searches for include files is:

1. The current directory if you use the "*filename*" construction.

   The current directory is not searched if you use the **<*filename*>** syntax.

2. The path that is specified with the assembler option **--include-directory**.

3. The path that is specified in the environment variable ASTCINC when the product was installed.

4. The default include directory in the installation directory.

The assembler does not allow a label with this directive.

Example

```
.INCLUDE 'storage\mem.asm'    ; include file
.INCLUDE <data.asm>           ; Do not look in
                              ; current directory
```

.LOCAL

Syntax

```
.LOCAL symbol[,symbol]...
```

Description

All symbols or labels defined in the current section or module are local to the module by default. You can change this default behavior with assembler option **--symbol-scope=global**.

With the `.LOCAL` directive you declare one of more symbols as local. It means that the specified symbols are explicitly local to the module in which you define them.

If the symbols that appear in the operand field are not used in the module, the assembler gives a warning.

The assembler does not allow a label with this directive.

Example

```
        .SDECL   ".data.io",DATA
        .SECT    ".data.io"
        .LOCAL   LOOPA   ; LOOPA is local to this section

LOOPA .HALF    0x100   ; assigns the value 0x100 to LOOPA
```

Related Information

.EXTERN (Import global section symbol)

.GLOBAL (Declare global section symbol)

.MACRO, .ENDM

Syntax

```
macro_name .MACRO [argument[,argument]...]
    ...
    macro_definition_statements
    ...
    .ENDM
```

Description

With the .MACRO directive you define a macro. Macros provide a shorthand method for handling a repeated pattern of code or group of instructions. You can define the pattern as a macro, and then call the macro at the points in the program where the pattern would repeat.

The definition of a macro consists of three parts:

• *Header*, which assigns a name to the macro and defines the arguments (.MACRO directive).

• *Body*, which contains the code or instructions to be inserted when the macro is called.

• *Terminator*, which indicates the end of the macro definition (.ENDM directive).

The arguments are symbolic names that the macro processor replaces with the literal arguments when the macro is expanded (called). Each formal *argument* must follow the same rules as symbol names: the name can consist of letters, digits and underscore characters (_). The first character cannot be a digit. Argument names cannot start with a percent sign (**%**).

Macro definitions can be nested but the nested macro will not be defined until the primary macro is expanded.

You can use the following operators in macro definition statements:

Operator	Name	Description
\	Macro argument concatenation	Concatenates a macro argument with adjacent alphanumeric characters.
?	Return decimal value of symbol	Substitutes the **?**_symbol_ sequence with a character string that represents the decimal value of the symbol.
%	Return hex value of symbol	Substitutes the **%**_symbol_ sequence with a character string that represents the hexadecimal value of the symbol.
"	Macro string delimiter	Allows the use of macro arguments as literal strings.
^	Macro local label override	Prevents name mangling on labels in macros.

Example

The macro definition:

```
CONST.D .MACRO  dx,v                            ;header
        movh    dx,#@his(v)                     ;body
```

```
        addi    dx,dx,#@los(v)
        .ENDM                                    ;terminator
```

The macro call:

```
    .SDECL    ".text",code
    .SECT     ".text"
    CONST.D  d4,0x12345678
```

The macro expands as follows:

```
    movh   d4,#@his(0x12345678)
    addi   d4,d4,#@los(0x12345678)
```

Related Information

Section 3.10, *Macro Operations*

.DUP, .ENDM (Duplicate sequence of source lines)

.DUPA, .ENDM (Duplicate sequence with arguments)

.DUPC, .ENDM (Duplicate sequence with characters)

.DUPF, .ENDM (Duplicate sequence in loop)

.PMACRO (Undefine macro)

.DEFINE (Define a substitution string)

.MESSAGE

Syntax

```
.MESSAGE {str|exp}[,{str|exp}]...
```

Description

With the `.MESSAGE` directive you tell the assembler to print a message to `stderr` during the assembling process.

An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be specified to describe the nature of the generated message. If you use expressions, the assembler outputs the result. The assembler outputs a space between each argument.

The error and warning counts will not be affected. The `.MESSAGE` directive is for example useful in combination with conditional assembly to indicate which part is assembled. The assembling process proceeds normally after the message has been printed.

This directive has no effect on the exit code of the assembler.

A label is not allowed with this directive.

Example

```
.DEFINE LONG "SHORT"
.MESSAGE 'This is a LONG string'
.MESSAGE "This is a LONG string"
```

Within single quotes, the defined symbol `LONG` is not expanded. Within double quotes the symbol `LONG` is expanded so the actual message is printed as:

```
This is a LONG string
This is a SHORT string
```

Related Information

`.FAIL` (Programmer generated error)

`.WARNING` (Programmer generated warning)

.MISRAC

Syntax

> **.MISRAC** *string*

Description

The C compiler can generate the .MISRAC directive to pass the compiler's MISRA C settings to the object file. The linker performs checks on these settings and can generate a report. It is not recommended to use this directive in hand-coded assembly.

Example

```
.MISRAC 'MISRA-C:2004,64,e2,0b,e,e11,27,6,ef83,e1,
        ef,66,cb75,af1,eff,e7,e7f,8d,63,87ff7,6ff3,4'
```

Related Information

Section 4.7.2, *C Code Checking: MISRA C*

C compiler option **--misrac**

.NAME

Syntax

```
.NAME string
```

Description

With the .NAME directive you specify the name of the original C source module. This directive is generated by the C compiler. You do not need this directive in hand-written assembly.

Example

```
.NAME "main.c"
```

.ORG

Syntax

.ORG [*abs-loc*][,*sect_type*][,*attribute*]...

Description

With the .ORG directive you can specify an absolute location (*abs_loc*) in memory of a section. This is the same as a .SDECL/.SECT without a section name.

This directive uses the following arguments:

abs-loc	Initial value to assign to the run-time location counter. *abs-loc* must be an absolute expression. If *abs_loc* is not specified, then the value is zero.
sect_type	An optional section type: code or data
attribute	An optional section attribute: init, noread, noclear, max, rom, group(*string*), cluster(*string*), protect

For more information about the section types and attributes see the assembler directive .SDECL.

The section type and attributes are case insensitive. A label is not allowed with this directive.

Example

```
; define a section at location 100 decimal
  .org   100

; define a relocatable nameless section
  .org

; define a relocatable data section
  .org   ,data

; define a data section at 0x8000
  .org   0x8000,data
```

Related Information

.SDECL (Declare section name and attributes)

.SECT (Activate a declared section)

.PMACRO

Syntax

```
.PMACRO symbol[,symbol]...
```

Description

With the .PMACRO directive you tell the assembler to undefine the specified macro, so that later uses of the symbol will not be expanded.

The assembler does not allow a label with this directive.

Example

```
.PMACRO MAC1,MAC2
```

This statement causes the macros named MAC1 and MAC2 to be undefined.

Related Information

.MACRO, .ENDM (Define a macro)

.SDECL

```
.SDECL "name",type[,attribute]... [AT address]
```

Description

With the `.SDECL` directive you can define a section with a *name*, *type* and optional *attributes*. Before any code or data can be placed in a section, you must use the `.SECT` directive to activate the section.

The *name* specifies the name of the section. The *type* operand specifies the section's type and must be one of:

Type	Description
CODE	Code section.
DATA	Data section.

The section type and attributes are case insensitive.

The defined *attribute*s are:

Attribute	Description	Allowed on type
AT *address*	Locate the section at the given *address*.	CODE, DATA
CLEAR	Sections are zeroed at startup.	DATA
CLUSTER('*name*')	Cluster code sections with companion debug sections. Used by the linker during removal of unreferenced sections. The name must be unique for this module (not for the application).	CODE, DATA
CONCAT	Concatenate sections. Used by the linker to merge sections with the same name.	CODE, DATA
INIT	Defines that the section contains initialization data, which is copied from ROM to RAM at program startup.	CODE, DATA
LINKONCE '*tag*'	For internal use only.	
MAX	When data sections with the same name occur in different object modules with the MAX attribute, the linker generates a section of which the size is the maximum of the sizes in the individual object modules.	DATA
NOCLEAR	Sections are not zeroed at startup. This is a default attribute for data sections. This attribute is only useful with BSS sections, which are cleared at startup by default.	DATA
NOREAD	Defines that the section can be executed from but not read.	CODE
PROTECT	Tells the linker to exclude a section from unreferenced section removal and duplicate section removal.	CODE, DATA
ROM	Section contains data to be placed in ROM. This ROM area is not executable.	CODE, DATA

Section names

The *name* of a section can have a special meaning for locating sections. The name of code sections should always start with ".`text`". With data sections, the prefix in the name is important. The prefix determines if the section is initialized, constant or uninitialized and which addressing mode is used. See the following table.

Name prefix	Type of section
.bss	uninitialized data
.bss_a0	uninitialized data, a0 addressing
.bss_a1	uninitialized data, a1 addressing
.bss_a8	uninitialized data, a8 addressing
.bss_a9	uninitialized data, a9 addressing
.data	initialized data
.data_a0	initialized data, a0 addressing
.data_a1	initialized data, a1 addressing
.data_a8	initialized data, a8 addressing
.data_a9	initialized data, a9 addressing
.ldata	constant data, a1 addressing (read only constants, literal data)
.rodata	constant data
.rodata_a0	constant data, a0 addressing
.rodata_a1	constant data, a1 addressing
.rodata_a8	constant data, a8 addressing
.rodata_a9	constant data, a9 addressing
.sbss	uninitialized data, a0 addressing
.sdata	initialized data, a0 addressing
.text	program code
.zbss	uninitialized data, abs 18 addressing
.zdata	initialized data, abs 18 addressing
.zrodata	constant data, abs 18 addressing

Note that the compiler uses the following name convention:

```
prefix.module_name.function_or_object_name
```

Also note that you cannot use the @ sign in section names. The assembler strips the @ sign and any following characters from the section name.

Example

```
  .sdecl  ".text.t.main", CODE   ; declare code section
  .sect   ".text.t.main"         ; activate section
```

```
.sdecl  ".data.t.var1", DATA   ; declare data section
.sect   ".data.t.var1"         ; activate section

.sdecl  ".text.intvec.00a", CODE   ; declare interrupt
                ; vector table entry for interrupt 10
.sect   ".text.intvec.00a"         ; activate section

.sdecl  ".data.t.abssec",data at 0x100
                                ; absolute section
.sect   ".data.t.abssec"       ; activate section
```

Related Information

.SECT (Activate a declared section)

.ORG (Initialize a nameless section)

.SECT

Syntax

.SECT "*name*" [**,RESET**]

Description

With the .SECT directive you activate a previously declared section with the name *name*. Before you can activate a section, you must define the section with the .SDECL directive. You can activate a section as many times as you need.

With the attribute RESET you can reset counting storage allocation in data sections that have section attribute MAX.

Example

```
.sdecl  ".zdata.t.var2", DATA   ; declare data section
.sect   ".zdata.t.var2"         ; activate section
```

Related Information

.SDECL (Declare section name and attributes)

.ORG (Initialize a nameless section)

.SET

Syntax

symbol **.SET** *expression*

.SET *symbol expression*

Description

With the .SET directive you assign the value of *expression* to symbol *temporarily*. If a symbol was defined with the .SET directive, you can redefine that symbol in another part of the assembly source, using the .SET directive again. Symbols that you define with the .SET directive are always local: you cannot define the symbol global with the .GLOBAL directive.

The .SET directive is useful in establishing temporary or reusable counters within macros. *expression* must be absolute and forward references are allowed.

Example

```
COUNT  .SET  0   ; Initialize count. Later on you can
                 ; assign other values to the symbol
```

Related Information

.EQU (Set permanent value to a symbol)

.SIZE

Syntax

```
.SIZE   symbol,expression
```

Description

With the `.SIZE` directive you set the size of the specified *symbol* to the value represented by *expression*.

The `.SIZE` directive may occur anywhere in the source file unless the specified symbol is a function. In this case, the `.SIZE` directive must occur after the function has been defined.

Example

```
main:  .type   func
       .        ; function main
       .
       ret16
main_function_end:
       .size   main,main_function_end-main
```

Related Information

`.TYPE` (Set symbol type)

.SPACE

Syntax

[*label*:] **.SPACE** *expression*

Description

The .SPACE directive reserves a block in memory. The reserved block of memory is not initialized to any value.

If you specify the optional *label*, it gets the value of the location counter at the start of the directive processing.

The *expression* specifies the number of MAUs (Minimal Addressable Units) to be reserved, and how much the location counter will advance. The expression must evaluate to an integer greater than zero and cannot contain any forward references (symbols that have not yet been defined). For the TriCore the MAU size is 8 (1 byte).

If you specify *label*, it gets the value of the location counter at the start of the directive processing.

Example

To reserve 12 bytes (not initialized) of memory in a TriCore data section:

```
        .sdecl  ".zbss.tst.uninit",DATA
        .sect   ".zbss.tst.uninit"
uninit  .SPACE  12     ; Sample buffer
```

Related Information

.BYTE (Define a constant byte)

.TYPE

Syntax

```
symbol  .TYPE  typeid
```

Description

With the `.TYPE` directive you set a *symbol*'s type to the specified value in the ELF symbol table. Valid symbol types are:

FUNC   The symbol is associated with a function or other executable code.

OBJECT   The symbol is associated with an object such as a variable, an array, or a structure.

FILE   The symbol name represents the filename of the compilation unit.

Labels in code sections have the default type FUNC. Labels in data sections have the default type OBJECT.

Example

```
Afunc:  .type   func
```

Related Information

`.SIZE` (Set symbol size)

.UNDEF

Syntax

.UNDEF *symbol*

Description

With the .UNDEF directive you can undefine a substitution string that was previously defined with the .DEFINE directive. The substitution string associated with *symbol* is released, and *symbol* will no longer represent a valid .DEFINE substitution or macro.

The assembler issues a warning if you redefine an existing symbol.

The assembler does not allow a label with this directive.

Example

The following example undefines the LEN substitution string that was previously defined with the .DEFINE directive:

```
.UNDEF LEN
```

Related Information

.DEFINE (Define a substitution string)

.WARNING

Syntax

 .WARNING {*str*|*exp*}[,{*str*|*exp*}]...

Description

With the .WARNING directive you tell the assembler to print a warning message to stderr during the assembling process.

An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be specified to describe the nature of the generated warning. If you use expressions, the assembler outputs the result. The assembler outputs a space between each argument.

The total warning count will be incremented as with any other warning. The .WARNING directive is for example useful in combination with conditional assembly to indicate which part is assembled. The assembling process proceeds normally after the message has been printed.

This directive has no effect on the exit code of the assembler, unless you use the assembler option **--warnings-as-errors**. In that case the assembler exits with exit code 1 (an error).

A label is not allowed with this directive.

Example

 .WARNING 'Parameter out of range'

This results in the warning:

W144: ["*filename*" *line*] Parameter out of range

Related Information

.FAIL (Programmer generated error)

.MESSAGE (Programmer generated message)

.WEAK

Syntax

```
.WEAK symbol[,symbol]...
```

Description

With the `.WEAK` directive you mark one or more symbols as 'weak'. The *symbol* can be defined in the same module with the `.GLOBAL` directive or the `.EXTERN` directive. If the symbol does not already exist, it will be created.

A 'weak' external reference is resolved by the linker when a global (or weak) definition is found in one of the object files. However, a weak reference will not cause the extraction of a module from a library to resolve the reference.

You can overrule a weak definition with a `.GLOBAL` definition in another module. The linker will not complain about the duplicate definition, and ignore the weak definition.

Only program labels and symbols defined with `.EQU` can be made weak.

Example

```
LOOPA .EQU 1          ; definition of symbol LOOPA
      .GLOBAL  LOOPA  ; LOOPA will be globally
                      ; accessible by other modules
      .WEAK LOOPA     ; mark symbol LOOPA as weak
```

Related Information

`.EXTERN` (Import global section symbol)

`.GLOBAL` (Declare global section symbol)

.WORD, .HALF

Syntax

```
[label:] .WORD argument[,argument]...
[label:] .HALF argument[,argument]...
```

Description

With the `.WORD` or `.HALF` directive the assembler allocates and initializes one word (32 bits) or a halfword (16 bits) of memory for each *argument*.

If you specify the optional *label*, it gets the value of the location counter at the start of the directive processing.

An *argument* can be a single- or multiple-character string constant, an expression or empty.

Multiple arguments are stored in sets of four or two bytes. One or more arguments can be null (indicated by two adjacent commas), in which case the corresponding byte location will be filled with zeros.

The value of the arguments must be in range with the size of the directive; floating-point numbers are not allowed. If the evaluated argument is too large to be represented in a word / halfword, the assembler issues a warning and truncates the value.

String constants

Single-character strings are stored in the most significant byte of a word / halfword, where the lower seven bits in that byte represent the ASCII value of the character, for example:

```
.WORD 'R'        ; = 0x52000000
.HALF 'R'        ; = 0x5200
```

Multiple-character strings are stored in consecutive byte addresses, as shown below. The standard C language escape characters like '\n' are permitted.

```
.WORD  'ABCD'          ; = 0x44434241
```

Example

When a string is supplied as argument of a directive that initializes multiple bytes, each character in the string is stored in consecutive bytes whose lower seven bits represent the ASCII value of the character. For example:

```
HTBL: .HALF 'ABC',,'D'   ; results in 0x424100004400 , the 'C' is truncated
WTBL: .WORD 'ABC'        ; results in 0x43424100
```

Related Information

`.BYTE` (Define a constant byte)

`.SPACE` (Define Storage)

3.9.2. Assembler Controls

Controls start with a **$** as the first character on the line. Unknown controls are ignored after a warning is issued.

Overview of assembler listing controls

Control	Description
`$LIST ON/OFF`	Print / do not print source lines to list file
`$PAGE`	Generate form feed in list file
`$PAGE` *settings*	Define page layout for assembly list file
`$PRCTL`	Send control string to printer
`$STITLE`	Set program subtitle in header of assembly list file
`$TITLE`	Set program title in header of assembly list file

Overview of miscellaneous assembler controls

Control	Description
`$CASE ON/OFF`	Case sensitive user names ON/OFF
`$CPU_TC`*num* `ON`	Enable/disable assembler check for specified functional problem
`$DEBUG ON/OFF`	Generation of symbolic debug ON/OFF
`$DEBUG "`*flags*`"`	Select debug information
`$HW_ONLY`	Prevent substitution of assembly instructions by smaller or faster instructions
`$IDENT LOCAL/GLOBAL`	Assembler treats labels by default as local or global
`$MMU`	Allow memory management instructions
`$NO_FPU`	Do not allow single precision floating-point instructions
`$OBJECT`	Alternative name for the generated object file
`$TC131` / `$TC16` / `$TC16X` / `$TC162`	Allow TriCore 1.3.1, 1.6,1.6.x or 1.6.2 instructions
`$WARNING OFF` [*num*]	Suppress all or some warnings

$CASE

Syntax

```
$CASE   ON
$CASE   OFF
```

Default

```
$CASE ON
```

Description

With the $CASE ON and $CASE OFF controls you specify wether the assembler operates in case sensitive mode or not. By default the assembler operates in case sensitive mode. This means that all user-defined symbols and labels are treated case sensitive, so LAB and Lab are distinct.

Note that the instruction mnemonics, register names, directives and controls are always treated case insensitive.

Example

```
;begin of source
$CASE OFF   ; assembler in case insensitive mode
```

Related Information

Assembler option **--case-insensitive**

$CPU_TCnum

Syntax

$CPU_TC*num* **ON**
$CPU_TC*num* **OFF**

Description

With these controls you can enable or disable specific CPU functional problem checks.

When you use this control, the define __CPU_TC*num*__ is set to 1.

Example

```
$CPU_TC018 ON   ; enable assembler check for CPU
                ; functional problem CPU_TC.018,
                ; __CPU_TC018__ is defined
```

Related Information

Assembler option **--silicon-bug**

Chapter 18, *CPU Problem Bypasses and Checks*

$DEBUG

Syntax

```
$DEBUG  ON
$DEBUG  OFF
$DEBUG  "flags"
```

Default

```
$DEBUG "AhLS"
```

Description

With the $DEBUG ON and $DEBUG OFF controls you turn the generation of debug information on or off. ($DEBUG ON is similar to the assembler option **--debug-info=+local** (**-gl**).

If you use the $DEBUG control with flags, you can set the following flags:

a/A	Assembly source line information
h/H	Pass high level language debug information (HLL)
l/L	Assembler local symbols debug information
s/S	Smart debug information

You cannot specify $DEBUG "ah". Either the assembler generates assembly source line information, or it passes HLL debug information.

Debug information that is generated by the C compiler, is always passed to the object file.

Example

```
;begin of source
$DEBUG ON   ; generate local symbols debug information
```

Related Information

Assembler option **--debug-info**

$HW_ONLY

Syntax

`$HW_ONLY`

Description

Normally the assembler replaces instructions by other, smaller or faster instructions. For example, the instruction `jeq d0,#0,label1` is replaced by `jz d0,label1`.

With the `$HW_ONLY` control you instruct the assembler to encode all instruction as they are. The assembler does not substitute instructions with other, faster or smaller instructions.

Example

```
;begin of source
$HW_ONLY   ; the assembler does not substitute
           ; instructions with other, smaller or
           ; faster instructions.
```

Related Information

Assembler option **--optimize=+generics**

$IDENT

Syntax

```
$IDENT LOCAL
$IDENT GLOBAL
```

Default

```
$IDENT LOCAL
```

Description

With the controls `$IDENT LOCAL` and `$IDENT GLOBAL` you tell the assembler how to treat symbols that you have not specified explicitly as local or global with the assembler directives `.LOCAL` or `.GLOBAL`.

By default the assembler treats all symbols as local symbols unless you have defined them to be global explicitly.

Example

```
;begin of source
$IDENT GLOBAL  ; assembly labels are global by default
```

Related Information

Assembler directive **.GLOBAL**

Assembler directive **.LOCAL**

Assembler option **--symbol-scope**

$LIST ON/OFF

Syntax

```
$LIST ON
$LIST OFF
```

Default

```
$LIST ON
```

Description

If you generate a list file with the assembler option **--list-file**, you can use the $LIST ON and $LIST OFF controls to specify which source lines the assembler must write to the list file. Without the assembler option **--list-file** these controls have no effect. The controls take effect starting at the next line.

The $LIST ON control actually increments a counter that is checked for a positive value and is symmetrical with respect to the $LIST OFF control. Note the following sequence:

```
; Counter value currently 1
$LIST ON        ; Counter value = 2
$LIST ON        ; Counter value = 3
$LIST OFF       ; Counter value = 2
$LIST OFF       ; Counter value = 1
```

The listing still would not be disabled until another $LIST OFF control was issued.

Example

```
   ...  ; source line in list file
$LIST OFF
   ...  ; source line not in list file
$LIST ON
   ...  ; source line also in list file
```

Related Information

Assembler option **--list-file**

Assembler function **@LST()**

$MMU

Syntax

`$MMU`

Description

With the `$MMU` control you instruct the assembler to accept and encode memory management instructions in the assembly source file.

When you use this control, the define `__MMU__` is set to 1.

Example

```
;begin of source
$MMU    ; the use of memory management instructions
        ; in this source is allowed.
```

Related Information

Assembler option **--mmu-present**

$NO_FPU

Syntax

`$NO_FPU`

Description

By default, the assembler accepts and encodes single precision floating-point (FPU) instructions in the assembly source file. With the `$NO_FPU` control you tell the assembler that FPU instructions are not allowed in the assembly source file.

When you use this control, the define `__FPU__` is set to 0. By default the define `__FPU__` is set to 1 which tells the assembler to accept single precision floating-point instructions.

Example

```
;begin of source
$NO_FPU    ; the use of single precision FPU instructions
           ; in this source is not allowed.
```

Related Information

Assembler option **--no-fpu**

$OBJECT

Syntax

```
$OBJECT "file"
$OBJECT OFF
```

Default

```
$OBJECT
```

Description

With the $OBJECT control you can specify an alternative name for the generated object file. With the $OBJECT OFF control, the assembler does not generate an object file at all.

Example

```
;Begin of source
$object "x1.o"          ; generate object file x1.o
```

Related Information

Assembler option **--output**

$PAGE

Syntax

$PAGE [*pagewidth*[,*pagelength*[,*blankleft*[,*blanktop*[,*blankbtm*]]]]]

Default

$PAGE 132,72,0,0,0

Description

If you generate a list file with the assembler option **--list-file**, you can use the $PAGE control to format the generated list file.

The arguments may be any positive absolute integer expression, and must be separated by commas.

pagewidth	Number of columns per line. The default is 132, the minimum is 40.
pagelength	Total number of lines per page. The default is 72, the minimum is 10. As a special case, a page length of 0 turns off page breaks.
blankleft	Number of blank columns at the left of the page. The default is 0, the minimum is 0, and the maximum must maintain the relationship: *blankleft* < *pagewidth*.
blanktop	Number of blank lines at the top of the page. The default is 0, the minimum is 0 and the maximum must be a value so that (*blanktop* + *blankbtm*) ≤ (*pagelength* - 10).
blankbtm	Number of blank lines at the bottom of the page. The default is 0, the minimum is 0 and the maximum must be a value so that (*blanktop* + *blankbtm*) ≤ (*pagelength* - 10).

If you use the $PAGE control without arguments, it causes a 'formfeed': the next source line is printed on the next page in the list file. The $PAGE control itself is not printed.

Example

```
$PAGE           ; formfeed, the next source line is printed
                ; on the next page in the list file.

$PAGE 96        ; set page width to 96. Note that you can
                ; omit the last four arguments.

$PAGE ,,,3,3    ; use 3 line top/bottom margins.
```

Related Information

Assembler option **--list-file**

$PRCTL

Syntax

$PRCTL *exp|string[,exp|string]...*

Description

If you generate a list file with the assembler option **--list-file**, you can use the $PRCTL control to send control strings to the printer.

The $PRCTL control simply concatenates its arguments and sends them to the listing file (the control line itself is not printed unless there is an error).

You can specify the following arguments:

expr   A byte expression which may be used to encode non-printing control characters, such as ESC.

string An assembler string, which may be of arbitrary length, up to the maximum assembler-defined limits.

The $PRCTL control can appear anywhere in the source file; the assembler sends out the control string at the corresponding place in the listing file.

If a $PRCTL control is the last line in the last input file to be processed, the assembler insures that all error summaries, symbol tables, and cross-references have been printed before sending out the control string. In this manner, you can use a $PRCTL control to restore a printer to a previous mode after printing is done.

Similarly, if the $PRCTL control appears as the first line in the first input file, the assembler sends out the control string before page headings or titles.

Example

```
$PRCTL  $1B,'E'  ; Reset HP LaserJet printer
```

Related Information

Assembler option **--list-file**

$STITLE

Syntax

$STITLE "*string***"**

Default

$STITLE ""

Description

If you generate a list file with the assembler option **--list-file**, you can use the $STITLE control to specify the program subtitle which is printed at the top of all succeeding pages in the assembler list file below the title.

The specified subtitle is valid until the assembler encounters a new $STITLE control. By default, the subtitle is empty.

The $STITLE control itself will not be printed in the source listing.

If the page width is too small for the title to fit in the header, it will be truncated.

Example

```
$TITLE    'This is the title'
$STITLE   'This is the subtitle'
```

Related Information

Assembler option **--list-file**

Assembler control **$TITLE**

$TC131 / $TC16 / $TC16X / $TC162

Syntax

```
$TC131
$TC16
$TC16X
$TC162
```

Description

With the $TC131, $TC16, $TC16X or $TC162 control you instruct the assembler to accept and encode
TriCore 1.3.1, 1.6, 1.6.x or 1.6.2 instructions respectively in the assembly source file.

When you use one of these controls, the define __CORE_TC131__, __CORE_TC16__, __CORE_TC16X__
or __CORE_TC162__ is set to 1 respectively. When no control and no **--core** option is given, the default
core is TC1.3 and the define __CORE_TC13__ is set to 1.

Example

```
;begin of source
$TC162   ; the use of TriCore 1.6.2 instructions
         ; in this source is allowed.
```

Related Information

Assembler option **--core**

$TITLE

Syntax

$TITLE "*string***"**

Default

$TITLE ""

Description

If you generate a list file with the assembler option **--list-file**, you can use the $TITLE control to specify the program title which is printed at the top of each page in the assembler list file.

The specified title is valid until the assembler encounters a new $TITLE control. By default, the title is empty.

The $TITLE control itself will not be printed in the source listing.

If the page width is too small for the title to fit in the header, it will be truncated.

Example

```
$TITLE  'This is the title'
```

Related Information

Assembler option **--list-file**

Assembler control **$STITLE**

$WARNING OFF

Syntax

$WARNING OFF [*number*]

Default

All warnings are reported.

Description

This control allows you to disable all or individual warnings. The *number* argument must be a valid warning message number.

Example

```
$WARNING OFF       ; all warning messages are suppressed

$WARNING OFF 135  ; suppress warning message 135
```

Related Information

Assembler option **--no-warnings**

3.10. Macro Operations

Macros provide a shorthand method for inserting a repeated pattern of code or group of instructions. You can define the pattern as a macro, and then call the macro at the points in the program where the pattern would repeat.

Some patterns contain variable entries which change for each repetition of the pattern. Others are subject to conditional assembly.

When a macro is called, the assembler executes the macro and replaces the call by the resulting in-line source statements. 'In-line' means that all replacements act as if they are on the same line as the macro call. The generated statements may contain substitutable arguments. The statements produced by a macro can be any processor instruction, almost any assembler directive, or any previously-defined macro. Source statements resulting from a macro call are subject to the same conditions and restrictions as any other statements.

Macros can be nested. The assembler processes nested macros when the outer macro is expanded.

3.10.1. Defining a Macro

The first step in using a macro is to define it.

The definition of a macro consists of three parts:

- *Header*, which assigns a name to the macro and defines the arguments (`.MACRO` directive).

- *Body*, which contains the code or instructions to be inserted when the macro is called.

- *Terminator*, which indicates the end of the macro definition (`.ENDM` directive).

A macro definition takes the following form:

```
macro_name  .MACRO [argument[,argument]...]
    ...
    macro_definition_statements
    ...
    .ENDM
```

For more information on the definition see the description of the `.MACRO` directive.

The `.DUP`, `.DUPA`, `.DUPC`, and `.DUPF` directives are specialized macro forms to repeat a block of source statements. You can think of them as a simultaneous definition and call of an unnamed macro. The source statements between the `.DUP`, `.DUPA`, `.DUPC`, and `.DUPF` directives and the `.ENDM` directive follow the same rules as macro definitions.

3.10.2. Calling a Macro

To invoke a macro, construct a source statement with the following format:

```
[label] macro_name [argument[,argument]...]  [; comment]
```

where,

label	An optional label that corresponds to the value of the location counter at the start of the macro expansion.
macro_name	The name of the macro. This may not start in the first column.
argument	One or more optional, substitutable arguments. Multiple arguments must be separated by commas.
comment	An optional comment.

The following applies to macro arguments:

- Each argument must correspond one-to-one with the formal arguments of the macro definition. If the macro call does not contain the same number of arguments as the macro definition, the assembler issues a warning.

- If an argument has an embedded comma or space, you must surround the argument by single quotes (').

- You can declare a macro call argument as null in three ways:

  - enter delimiting commas in succession with no intervening spaces

    ```
    macroname ARG1,,ARG3 ; the second argument is a null argument
    ```

  - terminate the argument list with a comma, the arguments that normally would follow, are now considered null

    ```
    macroname ARG1,      ; the second and all following arguments are null
    ```

  - declare the argument as a null string

- No character is substituted in the generated statements that reference a null argument.

3.10.3. Using Operators for Macro Arguments

The assembler recognizes certain text operators within macro definitions which allow text substitution of arguments during macro expansion. You can use these operators for text concatenation, numeric conversion, and string handling.

Operator	Name	Description
\	Macro argument concatenation	Concatenates a macro argument with adjacent alphanumeric characters.
?	Return decimal value of symbol	Substitutes the **?**_symbol_ sequence with a character string that represents the decimal value of the symbol.
%	Return hex value of symbol	Substitutes the **%**_symbol_ sequence with a character string that represents the hexadecimal value of the symbol.
"	Macro string delimiter	Allows the use of macro arguments as literal strings.
^	Macro local label override	Prevents name mangling on labels in macros.

Example: Argument Concatenation Operator - \

Consider the following macro definition:

```
SWAP_MEM .MACRO REG1,REG2            ;swap memory contents
     LD.W  D0,[A\REG1]               ;use D0 as temp
     LD.W  D1,[A\REG2]               ;use D1 as temp
     ST.W  [A\REG1],D1
     ST.W  [A\REG2],D0
     .ENDM
```

The macro is called as follows:

```
     SWAP_MEM  0,1
```

The macro expands as follows:

```
     LD.W  D0,[A0]
     LD.W  D1,[A1]
     ST.W  [A0],D1
     ST.W  [A1],D0
```

The macro preprocessor substitutes the character '0' for the argument `REG1`, and the character '1' for the argument `REG2`. The concatenation operator (****) indicates to the macro preprocessor that the substitution characters for the arguments are to be concatenated with the character 'A'.

Without the '\' operator the macro would expand as:

```
     LD.W  D0,[AREG1]
     LD.W  D1,[AREG2]
     ST.W  [AREG1],D1
     ST.W  [AREG2],D0
```

which results in an assembler error (invalid operand).

Example: Decimal Value Operator - ?

Instead of substituting the formal arguments with the actual macro call arguments, you can also use the value of the macro call arguments.

Consider the following source code that calls the macro `SWAP_SYM` after the argument `AREG` has been set to 0 and `BREG` has been set to 1.

```
AREG .SET     0
BREG .SET     1
     SWAP_SYM  AREG,BREG
```

If you want to replace the arguments with the value of `AREG` and `BREG` rather than with the literal strings `'AREG'` and `'BREG'`, you can use the **?** operator and modify the macro as follows:

```
SWAP_SYM   .MACRO  REG1,REG2         ;swap memory contents
     LD.W  D0,_lab\?REG1             ;use D0 as temp
```

```
        LD.W  D1,_lab\?REG2               ;use D1 as temp
        ST.W  _lab\?REG1,D1
        ST.W  _lab\?REG2,D0
        .ENDM
```

The macro first expands as follows:

```
        LD.W  D0,_lab\?AREG
        LD.W  D1,_lab\?BREG
        ST.W  _lab\?AREG,D1
        ST.W  _lab\?BREG,D0
```

Then ?AREG is replaced by '0' and ?BREG is replaced by '1':

```
        LD.W  D0,_lab\1
        LD.W  D1,_lab\2
        ST.W  _lab\1,D1
        ST.W  _lab\2,D0
```

Because of the concatenation operator '\' the strings are concatenated:

```
        LD.W  D0,_lab1
        LD.W  D1,_lab2
        ST.W  _lab1,D1
        ST.W  _lab2,D0
```

Example: Hex Value Operator - %

The percent sign (**%**) is similar to the standard decimal value operator (**?**) except that it returns the hexadecimal value of a symbol.

Consider the following macro definition:

```
GEN_LAB   .MACRO  LAB,VAL,STMT
LAB\%VAL  STMT
     .ENDM
```

The macro is called after NUM has been set to 10:

```
NUM  .SET       10
     GEN_LAB    HEX,NUM,NOP
```

The macro expands as follows:

```
HEXA NOP
```

The %VAL argument is replaced by the character 'A' which represents the hexadecimal value 10 of the argument VAL.

Example: Argument String Operator - "

To generate a literal string, enclosed by single quotes ('), you must use the argument string operator (")
in the macro definition.

Consider the following macro definition:

```
STR_MAC    .MACRO   STRING
    .BYTE   "STRING"
    .ENDM
```

The macro is called as follows:

```
    STR_MAC   ABCD
```

The macro expands as follows:

```
    .BYTE    'ABCD'
```

Within double quotes `.DEFINE` directive definitions can be expanded. Take care when using constructions
with single quotes and double quotes to avoid inappropriate expansions. Since `.DEFINE` expansion
occurs before macro substitution, any `.DEFINE` symbols are replaced first within a macro argument string:

```
    .DEFINE LONG   'short'
STR_MAC    .MACRO   STRING
    .MESSAGE 'This is a LONG STRING'
    .MESSAGE "This is a LONG STRING"
    .ENDM
```

If the macro is called as follows:

```
    STR_MAC   sentence
```

it expands as:

```
    .MESSAGE 'This is a LONG STRING'
    .MESSAGE 'This is a short sentence'
```

Macro Local Label Override Operator - ^

If you use labels in macros, the assembler normally generates another unique name for the labels (such
as LAB__M_L000001).

The macro **^**-operator prevents name mangling on macro local labels.

Consider the following macro definition:

```
INIT   .MACRO   addr
LAB:  LD.W    D0,^addr
      .ENDM
```

The macro is called as follows:

```
LAB:
        INIT LAB
```

The macro expands as:

```
LAB__M_L000001: LD.W  D0,LAB
```

If you would have omitted the **^** operator, the macro preprocessor would choose another name for LAB because the label already exists. The macro would expand like:

```
LAB__M_L000001: LD.W  D0,LAB__M_L000001
```

Chapter 4. Using the C Compiler

This chapter describes the compilation process and explains how to call the C compiler.

The TASKING VX-toolset for TriCore under Eclipse can use the internal builder (default) or the TASKING makefile generator (external builder) to build your entire embedded project, from C source till the final ELF/DWARF object file which serves as input for the debugger.

Although in Eclipse you cannot run the C compiler separately from the other tools, this section discusses the options that you can specify for the C compiler.

On the command line it is possible to call the C compiler separately from the other tools. However, it is recommended to use the control program for command line invocations of the toolset (see Section 8.1, *Control Program*). With the control program it is possible to call the entire toolset with only one command line.

The C compiler takes the following files for input and output:
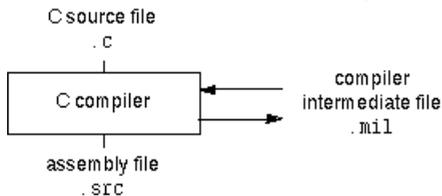
This chapter first describes the compilation process which consists of a *frontend* and a *backend* part. Next it is described how to call the C compiler and how to use its options. An extensive list of all options and their descriptions is included in Section 10.2, *C Compiler Options*. Finally, a few important basic tasks are described, such as including the C startup code and performing various optimizations.

4.1. Compilation Process

During the compilation of a C program, the C compiler runs through a number of phases that are divided into two parts: *frontend* and *backend*.

The backend part is not called for each C statement, but starts after a complete C module or set of modules has been processed by the frontend (in memory). This allows better optimization.

The C compiler requires only one pass over the input file which results in relative fast compilation.

Frontend phases

1. The preprocessor phase:

   The preprocessor includes files and substitutes macros by C source. It uses only string manipulations on the C source. The syntax for the preprocessor is independent of the C syntax but is also described in the ISO/IEC 9899:1999(E) standard.

2. The scanner phase:

   The scanner converts the preprocessor output to a stream of tokens.

3. The parser phase:

   The tokens are fed to a parser for the C grammar. The parser performs a syntactic and semantic analysis of the program, and generates an intermediate representation of the program. This code is called MIL (Medium level Intermediate Language).

4. The frontend optimization phase:

   Target processor independent optimizations are performed by transforming the intermediate code.

Backend phases

1. Instruction selector phase:

   This phase reads the MIL input and translates it into Low level Intermediate Language (LIL). The LIL objects correspond to a processor instruction, with an opcode, operands and information used within the C compiler.

2. Peephole optimizer/instruction scheduler/software pipelining phase:

   This phase replaces instruction sequences by equivalent but faster and/or shorter sequences, rearranges instructions and deletes unnecessary instructions.

3. Register allocator phase:

   This phase chooses a physical register to use for each virtual register.

4. The backend optimization phase:

   Performs target processor independent and dependent optimizations which operate on the Low level Intermediate Language.

5. The code generation/formatter phase:

   This phase reads through the LIL operations to generate assembly language output.

4.2. Calling the C Compiler

The TASKING VX-toolset for TriCore under Eclipse can use the internal builder (default) or the TASKING makefile generator (external builder) to build your entire project. After you have built your project, the output files are available in a subdirectory of your project directory, depending on the active configuration you have set in the **C/C++ Build » Settings** page of the **Project » Properties for** dialog.

Building a project under Eclipse

You have several ways of building your project:

- Build Selected File(s) (🔳). This compiles and assembles the selected file(s) without calling the linker.

    1. In the C/C++ Projects view, select the files you want to compile.

    2. Right-click in the C/C++ Projects view and select **Build Selected File(s)**.

- Build Individual Project (🔳).

    To build individual projects incrementally, select **Project » Build** *project*.

- Rebuild Project (🔳). This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.

    1. Select **Project » Clean...**

    2. Enable the option **Start a build immediately** and click **OK**.

- Build Automatically. This performs a build of all projects whenever any project file is saved, such as your makefile.

    This way of building is not recommended for C/C++ development, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item. In order for this option to work, you must also enable option **Build on resource save (Auto build)** on the **Behaviour** tab of the **C/C++ Build** page of the **Project » Properties for** dialog.

See also Chapter 11, *Influencing the Build Time*.

Select a target processor (core)

Processor options affect the invocation of all tools in the toolset. In Eclipse you only need to set them once. Based on the target processor, the compiler includes a special function register file. This is a regular include file which enables you to use virtual registers that are located in memory.

You can specify the target processor when you create a new project with the New C/C++ Project wizard (**File » New » TASKING TriCore C/C++ Project**), but you can always change the processor in the project properties dialog.

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Processor**.

    In the right pane the Processor page appears.

3. From the **Processor selection** list, select a processor.

To access the C/C++ compiler options

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler**.

4. Select the sub-entries and set the options in the various pages.

> Note that the C/C++ compiler options are used to create an object file from a C or C++ file. The options you enter in the Assembler page are not only used for hand-coded assembly files, but also for intermediate assembly files.

You can find a detailed description of all C compiler options in Section 10.2, *C Compiler Options*.

Invocation syntax on the command line (Windows Command Prompt):

`ctc` [[*option*]... [*file*]...]...

4.3. The C Startup Code

You need the run-time startup code to build an executable application. The startup code consists of the following components:

*   *Initialization code.* This code is executed when the program is initiated and before the function `main()` is called. It initializes the processor's registers and the application C variables.

*   *Exit code.* This controls the close down of the application after the program's main function terminates.

*   *Trap vector table.* This contains default trap vectors. See also Section 1.11.4, *Interrupt and Trap Functions*.

To add the C startup code to your project

When you create a new project with the New C/C++ Project wizard (**File » New » TASKING TriCore C/C++ Project**), fill in the dialogs and enable the option **Add startup file(s) to the project** in the following dialog (this is the default setting).
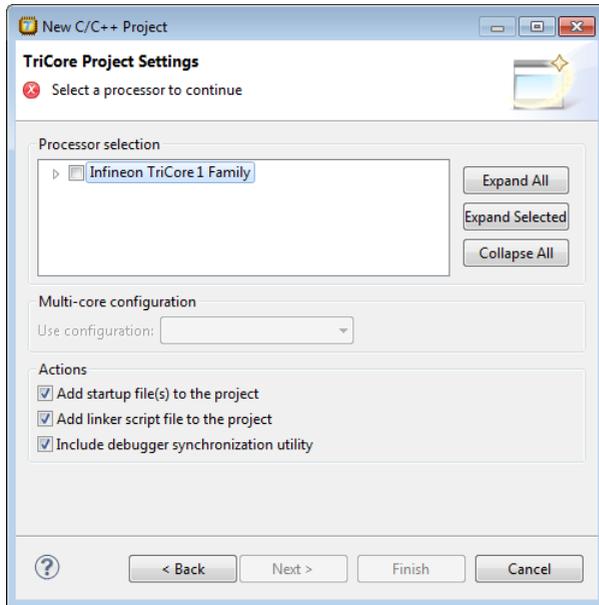
This adds the files `cstart.c` and `cstart.h` to your project. These files are copies of `lib/src/cstart.c` and `include/cstart.h`. If you do not add the startup code here, you can always add it later with **File » New » Startup Files**.

To change the C startup code configuration and registers

The project Properties dialog contains two pages where you can change the C startup code.

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Startup Configuration** or **Startup Registers**.

    In the right pane the Startup Configuration page or Startup Registers page appears.
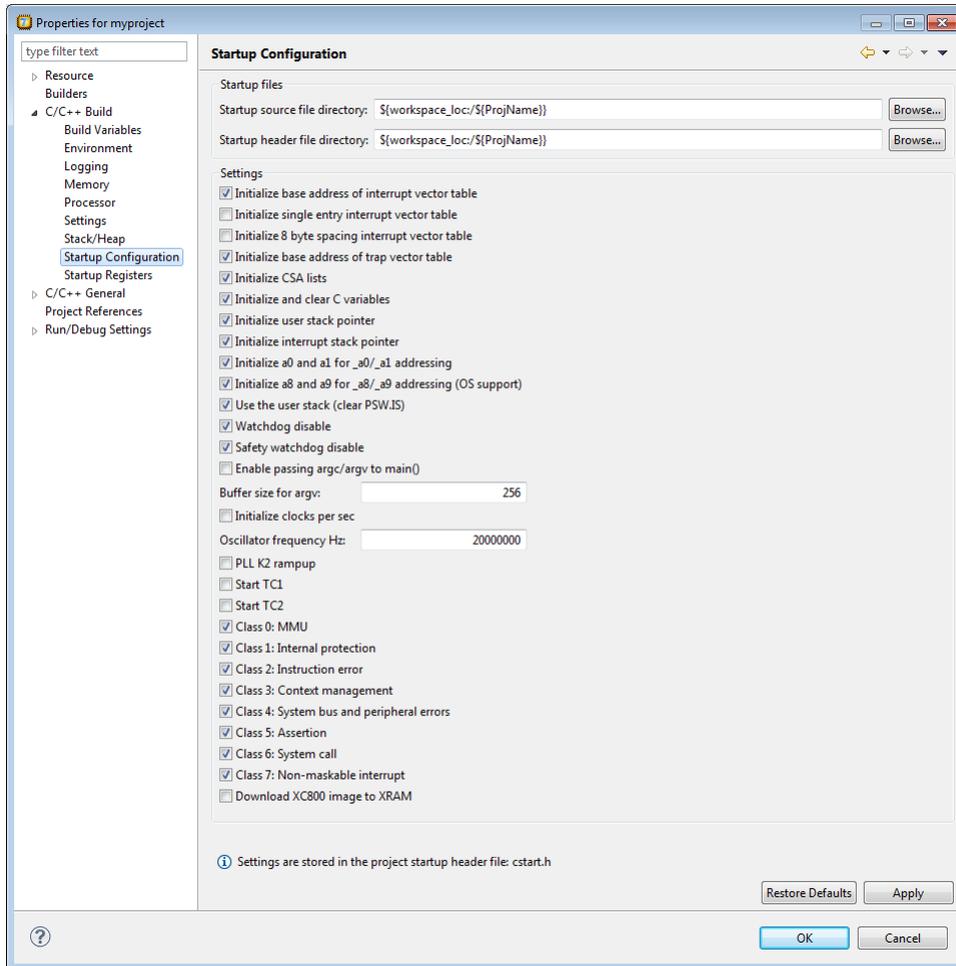
3. On the **Startup Configuration** page, you can make changes to the C startup code configuration.

4. On the **Startup Registers** page, you can specify the registers and their settings that must be known to the startup code. Enable the option **Initialize in startup code** to add a register setting to the startup code. If you made changes to a register and you want to reset the register to its original value, click on the **Default** button.

5. Click **OK**.

   The file cstart.h in your project is updated with the new values.

The values of the startup registers for a project are only set to their default values at project creation for the at that time selected processor.

When you switch to a different processor afterwards, in the **Project » Properties for » C/C++ Build » Processor** property page, the registers are not set to their defaults again. The reason for that is that you may have set specific values in the startup registers that you want to keep.

If you want to set all registers to their default values for the selected processor, you can do that any time by clicking on the **Restore Defaults** button on the **Project » Properties for » C/C++ Build » Startup Registers** property page.

When you use Import Board Configuration wizard to import (register) settings required for a certain board, only the registers needed to get the board going in the default situation are changed.

To change the C startup code in Eclipse manually

1. Double-click on the file `cstart.c` or `cstart.h`.

    The file opens in the editor area.

2. You can edit the C startup code directly in the editor.

    *A * appears in front of the name of the file to indicate that the file has changes.*

3. Click 🖫 or select **File » Save** to save the changes.

4.4. How the Compiler Searches Include Files

When you use include files (with the `#include` statement), you can specify their location in several ways. The compiler searches the specified locations in the following order:

1. If the `#include` statement contains an absolute pathname, the compiler looks for this file. If no path or a relative path is specified, the compiler looks in the same directory as the source file. This is only possible for include files that are enclosed in "".

   This first step is not done for include files enclosed in <>.

2. When the compiler did not find the include file, it looks in the directories that are specified in the **C/C++ Compiler » Include Paths** page in the **C/C++ Build » Settings » Tool Settings** tab of the Project Properties dialog (equivalent to the **-I** command line option).

3. When the compiler did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks in the path(s) specified in the environment variable `CTCINC`.

4. When the compiler still did not find the include file, it finally tries the default include directory relative to the installation directory (unless you specified option **--no-stdinc**).

Example

Suppose that the C source file `test.c` contains the following lines:

```
#include <stdio.h>
#include "myinc.h"
```

You can call the compiler as follows:

```
ctc -Imyinclude test.c
```

First the compiler looks for the file `stdio.h` in the directory `myinclude` relative to the current directory. If it was not found, the compiler searches in the environment variable `CTCINC` and then in the default `include` directory.

The compiler now looks for the file `myinc.h`, in the directory where `test.c` is located. If the file is not there the compiler searches in the directory `myinclude`. If it was still not found, the compiler searches in the environment variable `CTCINC` and then in the default `include` directory.

4.5. Compiling for Debugging

Compiling your files is the first step to get your application ready to run on a target. However, during development of your application you first may want to debug your application.

To create an object file that can be used for debugging, you must instruct the compiler to include symbolic debug information in the source file.

To include symbolic debug information

1. From the **Project** menu, select **Properties for**

   The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » Debugging**.

4. Select **Default** in the **Generate symbolic debug information** box.

Debug and optimizations

Due to different compiler optimizations, it might be possible that certain debug information is optimized away. Therefore, if you encounter strange behavior during debugging it might be necessary to reduce the optimization level, so that the source code is still suitable for debugging. For more information on optimization see Section 4.6, *Compiler Optimizations*.

Invocation syntax on the command line (Windows Command Prompt)

The invocation syntax on the command line is:

```
ctc -g file.c
```

4.6. Compiler Optimizations

The compiler has a number of optimizations which you can enable or disable.

1. From the **Project** menu, select **Properties for**

   The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » Optimization**.

4. Select an optimization level in the **Optimization level** box.

   or:

   In the **Optimization level** box select **Custom optimization** and enable the optimizations you want on the Custom optimization page.

Optimization levels

The TASKING C compiler offers four optimization levels and a custom level, at each level a specific set of optimizations is enabled.

- **Level 0 - No optimization**: No optimizations are performed. The compiler tries to achieve a 1-to-1 resemblance between source code and produced code. Expressions are evaluated in the order written in the source code, associative and commutative properties are not used.

- **Level 1 - Optimize**: Enables optimizations that do not affect the debug-ability of the source code. Use this level when you encounter problems during debugging your source code with optimization level 2.

- **Level 2 - Optimize more (default)**: Enables more optimizations to reduce the memory footprint and/or execution time. This is the default optimization level.

- **Level 3 - Optimize most**: This is the highest optimization level. Use this level when your program/hardware has become too slow to meet your real-time requirements.

- **Custom optimization**: you can enable/disable specific optimizations on the Custom optimization page.

Optimization pragmas

If you specify a certain optimization, all code in the module is subject to that optimization. Within the C source file you can overrule the C compiler options for optimizations with `#pragma optimize` *flag* and `#pragma endoptimize`. Nesting is allowed:

```
#pragma optimize e     /* Enable expression
...                       simplification            */
... C source ...
...
#pragma optimize c     /* Enable common expression
...                       elimination. Expression
... C source ...          simplification still enabled */
...
#pragma endoptimize    /* Disable common expression
...                       elimination               */
#pragma endoptimize    /* Disable expression
...                       simplification            */
```

The compiler optimizes the code between the pragma pair as specified.

You can enable or disable the optimizations described in the following subsection. The command line option for each optimization is given in brackets.

4.6.1. Generic Optimizations (frontend)

Common subexpression elimination (CSE) (option -Oc/-OC)

The compiler detects repeated use of the same (sub-)expression. Such a "common" expression is replaced by a variable that is initialized with the value of the expression to avoid recomputation. This method is called common subexpression elimination (CSE).

Expression simplification (option -Oe/-OE)

Multiplication by 0 or 1 and additions or subtractions of 0 are removed. Such useless expressions may be introduced by macros or by the compiler itself (for example, array subscripting).

Constant propagation (option -Op/-OP)

A variable with a known value is replaced by that value.

Automatic function inlining (option -Oi/-OI)

Small functions that are not too often called, are inlined. This reduces execution time at the cost of code size.

Control flow simplification (option -Of/-OF)

A number of techniques to simplify the flow of the program by removing unnecessary code and reducing the number of jumps. For example:

*   *Switch optimization*: A number of optimizations of a switch statement are performed, such as removing redundant case labels or even removing an entire switch.

*   *Jump chaining*: A (conditional) jump to a label which is immediately followed by an unconditional jump may be replaced by a jump to the destination label of the second jump. This optimization speeds up execution.

*   *Conditional jump reversal*: A conditional jump over an unconditional jump is transformed into one conditional jump with the jump condition reversed. This reduces both the code size and the execution time.

*   *Dead code elimination*: Code that is never reached, is removed. The compiler generates a warning messages because this may indicate a coding error.

Subscript strength reduction (option -Os/-OS)

An array or pointer subscripted with a loop iterator variable (or a simple linear function of the iterator variable), is replaced by the dereference of a pointer that is updated whenever the iterator is updated.

Loop transformations (option -Ol/-OL)

Transform a loop with the entry point at the bottom, to a loop with the entry point at the top. This enables constant propagation in the initial loop test and code motion of loop invariant code by the CSE optimization.

Forward store (option -Oo/-OO)

A temporary variable is used to cache multiple assignments (stores) to the same non-automatic variable.

Branch prediction (option -O-predict/-O+predict)

A prediction is done if branches are likely to be taken or not. Based on this, other optimizations can take place. Only with tradeoff level < 4 a branch is re-written with a larger instruction to improve performance. This option is not available for TriCore1.6, because TriCore1.6 supports dynamic branch prediction.

MIL linking (Control program option --mil-link)

The frontend phase performs its optimizations on the MIL code. When all C modules and/or MIL modules of an application are given to the C compiler in a single invocation, the C compiler will link MIL code of the modules to a complete application automatically. Next, the frontend will run its optimizations again with application scope. After this, the MIL code is passed on to the backend, which will generate a single `.src` file for the whole application. Linking with the run-time library, floating-point library and C library is still necessary. Linking with the C library is required because this library contains some hand-coded assembly functions, that are not linked in at MIL level.

In the ISO C99 standard a "translation unit" is a preprocessed source file together with all the headers and source files included via the preprocessing directive `#include`. After MIL linking the compiler will treat the linked sources files as a single translation unit, allowing global optimizations to be performed, that otherwise would be limited to a single module.
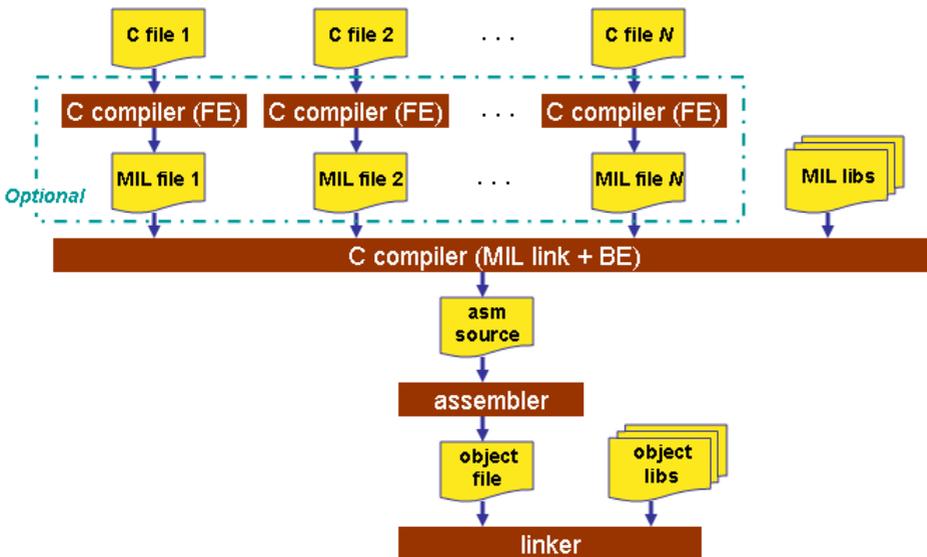
MIL splitting (option --mil-split)

When you specify that the C compiler has to use MIL splitting, the C compiler will first link the application at MIL level as described above. However, after rerunning the optimizations the MIL code is not passed on to the backend. Instead the frontend writes a `.ms` file for each input file or library. A `.ms` file has the same format as a `.mil` file. Only `.ms` files that really change are updated. The advantage of this approach is that it is possible to use the make utility to translate only those parts of the application to a `.src` file that really have changed. MIL splitting is therefore a more efficient build process than MIL linking. The

penalty for this is that the code compaction optimization in the backend does not have application scope. As with MIL linking, it is still required to link with the normal libraries to build an ELF file.
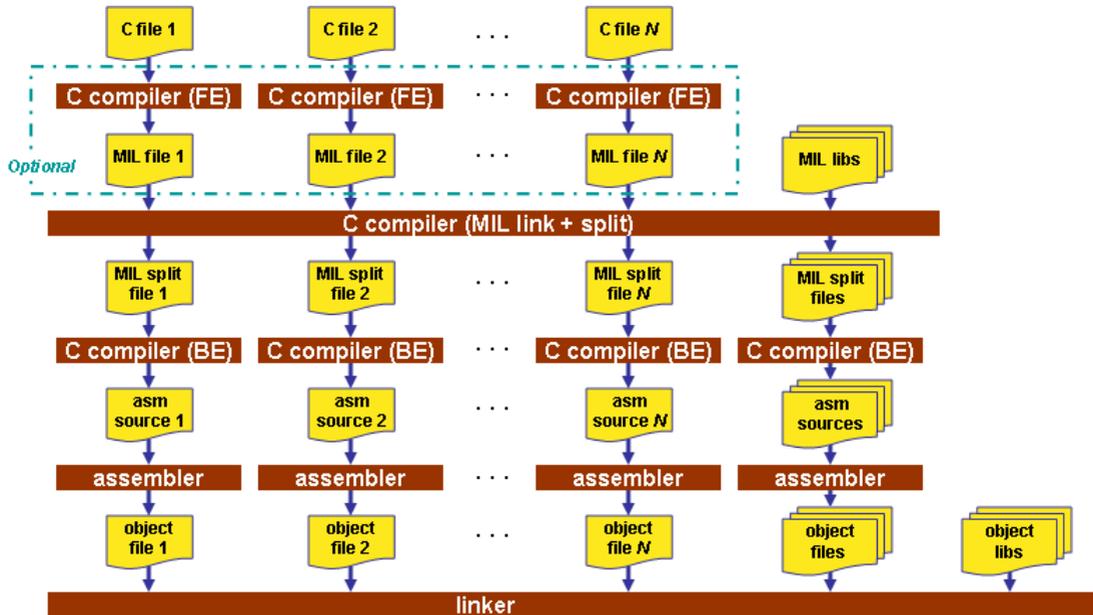
To read more about how MIL linking influences the build process of your application, see Section 11.2, *MIL Linking*.

4.6.2. Core Specific Optimizations (backend)

Coalescer (option -Oa/-OA)

The coalescer seeks for possibilities to reduce the number of moves (MOV instruction) by smart use of registers. This optimizes both speed and code size.

Peephole optimizations (option -Oy/-OY)

The generated assembly code is improved by replacing instruction sequences by equivalent but faster and/or shorter sequences, or by deleting unnecessary instructions.

Align loop bodies (option -On/-ON)

Loop bodies are aligned to lower the number of fetches required to retrieve the loop body.

The loop alignment for non-cached (external) memory in segment 0xa requires a different alignment than cached memory in segment 0x8.

Loops are only aligned when the loop alignment optimization option (this option) is enabled and the trade-off between speed and size is less than 3. (**--tradeoff=**{0..2}). This is because loop alignment is a speed optimization that will cost code size.

To reduce the number of fetches in non-cached memory, loops are best aligned on 256-bit, which is the length of a fetch line. Code size can increase significantly when aligning loops on 256-bit. Loop alignment for (external) non-cached memory is enabled with option **--loop=-cache** (only for TriCore 1.3 and 1.3.1).

Loop performance in cached memory cannot be improved efficiently with a simple 256-bit alignment. Instead of alignment with nops, 16-bit instructions before the loop body are replaced with 32-bit alternative instructions and the function starts at a 256-bit alignment so that instruction(s) that are at the loop start that have alignment requirements do not cross a 256-bit cache line. For example, dual issue instructions (i.e. instructions that can execute simultaneously) should not cross a 256-bit cache line, but they can be located at any location within a 256-bit offset. The loop entry label itself does not have to be at a 256-bit offset. No alignment is required when the instructions at the loop top do not have any cache alignment issue. For example, when the loop starts with a 16-bit single issue instruction. When performance is required, loops are best located in cached memory. With option **--loop=+cache** (only for TriCore 1.3 and 1.3.1; this option is the default) loops are aligned for cached memory or SPRAM.

Loop alignments are expected superfluous for TC1.6 performance, but you can enable a fixed loop alignment with option **--loop=+value**.

The loop alignment optimization of cached and non-cached memory is limited by loop characteristics. Only loops that are below the loop body instruction count threshold, size threshold or cycles threshold are aligned. These individual thresholds can be toggled with the loop control option **--loop=i/I,s/S,t/T**. The threshold values can be set with the options **--loop-cycle-threshold**, **--loop-instruction-threshold** and **--loop-size-threshold**.

The default alignment value is 32 bytes, because this is equal to a 256-bit fetch line. This alignment value can be overruled when **--loop=v** is enabled and another alignment value is defined with option **--loop-alignment**.

Instruction Scheduler (option -Ok/-OK)

The instruction scheduler is a backend optimization that acts upon the generated instructions. When two instructions need the same machine resource - like a bus, register or functional unit - at the same time, they suffer a *structural hazard*, which stalls the pipeline. This optimization tries to rearrange instructions to avoid structural hazards, for example by inserting another non-related instruction, or pairing a L/S instruction with a data arithmetic instruction in order to fill both pipelines as much as possible.

First the instruction stream is partitioned into basic blocks. A new basic block starts at a label, or right after a jump instruction. Unschedulable instructions and, when **-Av** is enabled, instructions that access volatile objects, each get their own basic block. Next, the scheduler searches the instructions within a basic block, looking for places where the pipeline stalls. After identifying these places it tries to rebuild the basic block using the existing instructions, while avoiding the pipeline stalls. In this process data dependencies between instructions are honoured.

Note that the function inlining optimization happens in the frontend of the compiler. The instruction scheduler has no knowledge about the origin of the instructions.

Unroll small loops (option -Ou/-OU)

To reduce the number of branches, short loops are eliminated by replacing them with a number of copies.

IFconversion (option -Ov/-OV)

IF - ELSE constructions are transformed into predicated instructions. This avoids unnecessary jumps while the predicated instructions are optimized by the *pipeline scheduler* and the predicate optimization.

Software pipelining (option -Ow/-OW)

A number of techniques to optimize loops. For example, within a loop the most efficient order of instructions is chosen by the *pipeline scheduler* and it is examined what instructions can be executed parallel.

Code compaction (reverse inlining) (option -Or/-OR)

Compaction is the opposite of inlining functions: chunks of code that occur more than once, are transformed into a function. This reduces code size at the cost of execution speed. The size of the chunks of code to be inlined depends on the setting of the C compiler option **--tradeoff** (**-t**). See the subsection **Code Compaction** in Section 4.6.3, *Optimize for Code Size or Execution Speed*.

Note that if you use section renaming, by default, the compiler only performs code compaction on sections that have the same section type prefix, core association and name given by the section renaming pragma or option. The module name and symbol name are not relevant in this case. When you use C compiler option **--relax-compact-name-check**, the compiler does not perform this section name check, but performs code compaction whenever possible.

Use of SIMD instructions (option -Om/-OM)

The iteration counts of loops are reduced where possible by taking advantage of the TriCore SIMD instructions. This optimizes speed, but may cause a slight increase in code size.

Generic assembly optimizations (option -Og/-OG)

A set of optimizations on the generated assembly code that increase speed and decrease code size, similar to peephole optimizations applied within and across basic blocks. The set includes but is not limited to:

• removal of unused code

• removal of superfluous code

• loop optimizations

• flow optimizations

• load/store optimizations

• addressing mode optimizations

4.6.3. Optimize for Code Size or Execution Speed

You can tell the compiler to focus on execution speed or code size during optimizations. You can do this by specifying a size/speed trade-off level from 0 (speed) to 4 (size). This trade-off does not turn optimization phases on or off. Instead, its level is a weight factor that is used in the different optimization phases to

influence the heuristics. The higher the level, the more the compiler focusses on code size optimization. To choose a trade-off value read the description below about which optimizations are affected and the impact of the different trade-off values.

Note that the trade-off settings are directions and there is no guarantee that these are followed. The compiler may decide to generate different code if it assessed that this would improve the result.

To specify the size/speed trade-off optimization level:

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » Optimization**.

4. Select a trade-off level in the **Trade-off between speed and size** box.

See also C compiler option **--tradeoff** (**-t**)

Instruction Selection

Trade-off levels 0, 1 and 2: the compiler selects the instructions with the smallest number of cycles.

Trade-off levels 3 and 4: the compiler selects the instructions with the smallest number of bytes.

Switch Jump Chain versus Jump Table

Instruction selection for the `switch` statements follows different trade-off rules. A switch statement can result in a jump chain or a jump table. The compiler makes the decision between those by measuring and weighing bytes and cycles. This weigh is controlled with the trade-off values:

Trade-off value	Time	Size
0	100%	0%
1	75%	25%
2	50%	50%
3	25%	75%
4	0%	100%

Loop Optimization

For a top-loop, the loop is entered at the top of the loop. A bottom-loop is entered at the bottom. Every loop has a test and a jump at the bottom of the loop, otherwise it is not possible to create a loop. Some top-loops also have a conditional jump before the loop. This is only necessary when the number of loop iterations is unknown. The number of iterations might be zero, in this case the conditional jump jumps over the loop.

Bottom loops always have an unconditional jump to the loop test at the bottom of the loop.

Trade-off value	Try to rewrite top-loops to bottom-loops	Optimize loops for size/speed
0	no	speed
1	yes	speed
2	yes	speed
3	yes	size
4	yes	size

Example:

```
int a;

void i( int l, int m )
{
    int i;

    for ( i = m; i < l; i++ )
    {
        a++;
    }
    return;
}
```

Coded as a bottom loop (compiled with **--tradeoff=4**) is:

```
        j    .L2            ;; unconditional jump to loop test at bottom
.L3:
        ld.w  d15,1
        add16 d15,#1
        st.w  a,d15
        add16 d5,#1
.L2:                        ;; loop entry point
        jlt   d5,d4,.L3
```

Coded as a top loop (compiled with **--tradeoff=0**) is:

```
        ld.w  d15,a        ;; test for at least one loop iteration
        jlt   d5,d4,.L2    ;; can be omitted when number of iterations is known
.L3:                       ;; loop entry point
        add16 d15,#1
        add16 d5,#1
        jlt   d5,d4,.L3
.L2:
        st.w  a,d15
```

Align Loop Bodies (-On)

Trade-off levels 0, 1 and 2: the compiler allows the align loop bodies optimization.

Trade-off levels 3 and 4: the compiler disables the align loop bodies optimization.

Automatic Function Inlining

You can enable automatic function inlining with the option **--optimize=+inline** (**-Oi**) or by using `#pragma optimize +inline`. This option is also part of the **-O3** predefined option set.

When automatic inlining is enabled, you can use the options **--inline-max-incr** and **--inline-max-size** (or their corresponding pragmas `inline_max_incr` / `inline_max_size`) to control automatic inlining. By default their values are set to -1. This means that the compiler will select a value depending upon the selected trade-off level. The defaults are:

Trade-off value	inline-max-incr	inline-max-size
0	100	50
1	50	25
2	20	20
3	10	10
4	0	0

For example with trade-off value 1, the compiler inlines all functions that are smaller or equal to 25 internal compiler units. After that the compiler tries to inline even more functions as long as the function will not grow more than 50%.

When these options/pragmas are set to a value >= 0, the specified value is used instead of the values from the table above.

Static functions that are called only once, are always inlined, independent of the values chosen for inline-max-incr and inline-max-size.

Code Compaction

Trade-off levels 0 and 1: code compaction is disabled.

Trade-off level 2: only code compaction of matches outside loops.

Trade-off level 3: code compaction of matches outside loops, and matches inside loops of patterns that have an estimate execution frequency lower or equal to 10.

Trade-off level 4: code compaction of matches outside loops, and matches inside loops of patterns that have an estimate execution frequency lower or equal to 100.

For loops where the iteration count is unknown an iteration count of 10 is assumed.

For the execution frequency the compiler also accounts nested loops.

See C compiler option **--compact-max-size**

4.7. Static Code Analysis

Static code analysis (SCA) is a relatively new feature in compilers. Various approaches and algorithms exist to perform SCA, each having specific pros and cons.

SCA Implementation Design Philosophy

SCA is implemented in the TASKING compiler based on the following design criteria:

• An SCA phase does not take up an excessive amount of execution time. Therefore, the SCA can be performed during a normal edit-compile-debug cycle.

• SCA is implemented in the compiler front-end. Therefore, no new makefiles or work procedures have to be developed to perform SCA.

• The number of emitted false positives is kept to a minimum. A false positive is a message that indicates that a correct code fragment contains a violation of a rule/recommendation. A number of warnings is issued in two variants, one variant when it is *guaranteed* that the rule is violated when the code is executed, and the other variant when the rules is *potentially* violated, as indicated by a preceding warning message.

For example see the following code fragment:

```
extern int some_condition(int);
void f(void)
{
    char buf[10];
    int i;

    for (i = 0; i <= 10; i++)
    {
        if (some_condition(i))
        {
            buf[i] = 0; /* subscript may be out of bounds */
        }
    }
}
```

As you can see in this example, if `i=10` the array `buf[]` might be accessed beyond its upper boundary, depending on the result of `some_condition(i)`. If the compiler cannot determine the result of this function at run-time, the compiler issues the warning "subscript is *possibly* out of bounds" preceding the CERT warning "ARR35: do not allow loops to iterate beyond the end of an array". If the compiler can determine the result, or if the `if` statement is omitted, the compiler can guarantee that the "subscript is out of bounds".

• The SCA implementation has real practical value in embedded system development. There are no real objective criteria to measure this claim. Therefore, the TASKING compilers support well known standards for safety critical software development such as the MISRA guidelines for creating software for safety critical automotive systems and secure "CERT C Secure Coding Standard" released by CERT. CERT is founded by the US government and studies internet and networked systems security vulnerabilities, and develops information to improve security.

Effect of optimization level on SCA results

The SCA implementation in the TASKING compilers has the following limitations:

- Some violations of rules will only be detected when a particular optimization is enabled, because they rely on the analysis done for that optimization, or on the transformations performed by that optimization. In particular, the constant propagation and the CSE/PRE optimizations are required for some checks. It is preferred that you enable these optimizations. These optimizations are enabled with the default setting of the optimization level (**-O2**).

- Some checks require cross-module inspections and violations will only be detected when multiple source files are compiled and linked together by the compiler in a single invocation.

4.7.1. C Code Checking: CERT C

The CERT C Secure Coding Standard provides rules and recommendations for secure coding in the C programming language. The goal of these rules and recommendations is to eliminate insecure coding practices and undefined behaviors that can lead to exploitable vulnerabilities. The application of the secure coding standard will lead to higher-quality systems that are robust and more resistant to attack.

For details about the standard, see the CERT C Secure Coding Standard web site. For general information about CERT secure coding, see www.cert.org/secure-coding.

Versions of the CERT C standard

Version 1.0 of the CERT C Secure Coding Standard is available as a book by Robert C. Seacord [Addison-Wesley]. Whereas the web site is a wiki and reflects the latest information, the book serves as a fixed point of reference for the development of compliant applications and source code analysis tools.

The rules and recommendations supported by the TASKING compiler reflect the version of the CERT web site as of June 1 2009.

The following rules/recommendations implemented by the TASKING compiler, are not part of the book: PRE11-C, FLP35-C, FLP36-C, MSC32-C

For a complete overview of the supported CERT C recommendations/rules by the TASKING compiler, see Chapter 19, *CERT C Secure Coding Standard*.

Priority and Levels of CERT C

Each CERT C rule and recommendation has an assigned *priority*. Three values are assigned for each rule on a scale of 1 to 3 for

- severity - how serious are the consequences of the rule being ignored

  1. low (denial-of-service attack, abnormal termination)

  2. medium (data integrity violation, unintentional information disclosure)

  3. high (run arbitrary code)

- likelihood - how likely is it that a flaw introduced by ignoring the rule could lead to an exploitable vulnerability

    1. unlikely

    2. probable

    3. likely

- remediation cost - how expensive is it to comply with the rule

    1. high (manual detection and correction)

    2. medium (automatic detection and manual correction)

    3. low (automatic detection and correction)

The three values are then multiplied together for each rule. This product provides a measure that can be used in prioritizing the application of the rules. These products range from 1 to 27. Rules and recommendations with a priority in the range of 1-4 are level 3 rules (low severity, unlikely, expensive to repair flaws), 6-9 are level 2 (medium severity, probable, medium cost to repair flaws), and 12-27 are level 1 (high severity, likely, inexpensive to repair flaws).

The TASKING compiler checks most of the level 1 and some of the level 2 CERT C recommendations/rules.

For a complete overview of the supported CERT C recommendations/rules by the TASKING compiler, see Chapter 19, *CERT C Secure Coding Standard*.

To apply CERT C code checking to your application

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » CERT C Secure Coding**.

4. Make a selection from the **CERT C secure code checking** list.

5. If you selected **Custom**, expand the **Custom CERT C** entry and enable one or more individual recommendations/rules.

On the command line you can use the option **--cert**.

```
ctc --cert={all | name [-name],...]
```

With **--diag=cert** you can see a list of the available checks, or you can use a three-letter mnemonic to list only the checks in a particular category. For example, **--diag=pre** lists all supported checks in the preprocessor category.

4.7.2. C Code Checking: MISRA C

The C programming language is a standard for high level language programming in embedded systems, yet it is considered somewhat unsuitable for programming safety-related applications. Through enhanced code checking and strict enforcement of best practice programming rules, TASKING MISRA C code checking helps you to produce more robust code.

MISRA C specifies a subset of the C programming language which is intended to be suitable for embedded automotive systems. It consists of a set of rules, defined in *MISRA-C:2004, Guidelines for the Use of the C Language in Critical Systems* (Motor Industry Research Association (MIRA), 2004).

The compiler also supports MISRA C:1998, the first version of MISRA C and MISRA C: 2012, the latest version of MISRA C. You can select the version with the following C compiler option:

```
--misrac-version=1998
--misrac-version=2004
--misrac-version=2012
```

In your C source files you can check against the MISRA C version used. For example:

```
#if __MISRAC_VERSION__ == 1998
  ...
#elif __MISRAC_VERSION__ == 2004
  ...
#elif __MISRAC_VERSION__ == 2012
  ...
#endif
```

For a complete overview of all MISRA C rules, see Chapter 20, *MISRA C Rules*.

Implementation issues

The MISRA C implementation in the compiler supports nearly all rules. Only a few rules are not supported because they address documentation, run-time behavior, or other issues that cannot be checked by static source code inspection, or because they require an application-wide overview.

During compilation of the code, violations of the enabled MISRA C rules are indicated with error messages and the build process is halted.

MISRA C rules are divided in mandatory rules, required rules and advisory rules. If rules are violated, errors are generated causing the compiler to stop. With the following options warnings, instead of errors, are generated:

```
--misrac-mandatory-warnings
--misrac-required-warnings
--misrac-advisory-warnings
```

> Note that not all MISRA C violations will be reported when other errors are detected in the input
> source. For instance, when there is a syntax error, all semantic checks will be skipped, including
> some of the MISRA C checks. Also note that some checks cannot be performed when the
> optimizations are switched off.

Quality Assurance report

To ensure compliance to the MISRA C rules throughout the entire project, the TASKING linker can
generate a MISRA C Quality Assurance report. This report lists the various modules in the project with
the respective MISRA C settings at the time of compilation. You can use this in your company's quality
assurance system to provide proof that company rules for best practice programming have been applied
in the particular project.

To apply MISRA C code checking to your application

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » MISRA C**.

4. Select the **MISRA C version** (1998, 2004 or 2012).

5. In the **MISRA C checking** box select a MISRA C configuration. Select a predefined configuration
    for conformance with the required rules in the MISRA C guidelines.

6. (Optional) In the **Custom 1998**, **Custom 2004** or **Custom 2012** entry, specify the individual rules.

```
ctc --misrac={all | number [-number],...]
```

4.8. C Compiler Error Messages

The C compiler reports the following types of error messages in the Problems view of Eclipse.

F (Fatal errors)

After a fatal error the compiler immediately aborts compilation.

E (Errors)

Errors are reported, but the compiler continues compilation. No output files are produced unless you have
set the C compiler option **--keep-output-files** (the resulting output file may be incomplete).

W (Warnings)

Warning messages do not result into an erroneous assembly output file. They are meant to draw your attention to assumptions of the compiler for a situation which may not be correct. You can control warnings in the **C/C++ Build » Settings » Tool Settings » C/C++ Compiler » Diagnostics** page of the **Project » Properties for** menu (C compiler option **--no-warnings**).

I (Information)

Information messages are always preceded by an error message. Information messages give extra information about the error.

S (System errors)

System errors occur when internal consistency checks fail and should never occur. When you still receive the system error message

```
S9##: internal consistency check failed - please report
```

please report the error number and as many details as possible about the context in which the error occurred.

Display detailed information on diagnostics

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

    The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

    A popup menu appears.

3. Select **Detailed Diagnostics Info**.

    A dialog box appears with additional information.

On the command line you can use the C compiler option **--diag** to see an explanation of a diagnostic message:

```
ctc --diag=[format:]{all | number,...]
```

Chapter 5. Using the C++ Compiler

This chapter describes the compilation process and explains how to call the C++ compiler. You should be familiar with the C++ language and with the ISO C language.

The C++ compiler can be seen as a preprocessor or front end which accepts C++ source files or sources using C++ language features. The output generated by the TriCore C++ compiler (**cptc**) is intermediate C, which can be translated with the TriCore C compiler (**ctc**).

The C++ compiler is part of a complete toolset, the TASKING VX-toolset for TriCore. For details about the C compiler see Chapter 4, *Using the C Compiler*.

The C++ compiler takes the following files for input and output:
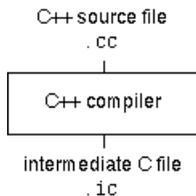
Although in Eclipse you cannot run the C++ compiler separately from the other tools, this section discusses the options that you can specify for the C++ compiler.

On the command line it is possible to call the C++ compiler separately from the other tools. However, it is recommended to use the control program for command line invocations of the toolset (see Section 8.1, *Control Program*). With the control program it is possible to call the entire toolset with only one command line. Eclipse also uses the control program to call the C++ compiler. Files with the extensions `.cc`, `.cpp` or `.cxx` are seen as C++ source files and passed to the C++ compiler.

The C++ compiler accepts the C++ language of the ISO/IEC 14882:2003 C++ standard, with some minor exceptions documented in Chapter 2, *C++ Language*. It also accepts embedded C++ language extensions.

The C++ compiler does no optimization. Its goal is to produce quickly a complete and clean parsed form of the source program, and to diagnose errors. It does complete error checking, produces clear error messages (including the position of the error within the source line), and avoids cascading of errors. It also tries to avoid seeming overly finicky to a knowledgeable C or C++ programmer.

5.1. Calling the C++ Compiler

Under Eclipse you cannot run the C++ compiler separately. However, you can set options specific for the C++ compiler. After you have built your project, the output files are available in a subdirectory of your project directory, depending on the active configuration you have set in the **C/C++ Build » Settings** page of the **Project » Properties for** dialog.

Building a project under Eclipse

You have several ways of building your project:

- Build Selected File(s) (🔲). This compiles and assembles the selected file(s) without calling the linker.

  1. In the C/C++ Projects view, select the files you want to compile.

  2. Right-click in the C/C++ Projects view and select **Build Selected File(s)**.

- Build Individual Project (🔲).

  To build individual projects incrementally, select **Project » Build** *project*.

- Rebuild Project (🔲). This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.

  1. Select **Project » Clean...**

  2. Enable the option **Start a build immediately** and click **OK**.

- Build Automatically. This performs a build of all projects whenever any project file is saved, such as your makefile.

  This way of building is not recommended for C/C++ development, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item. In order for this option to work, you must also enable option **Build on resource save (Auto build)** on the **Behaviour** tab of the **C/C++ Build** page of the **Project » Properties for** dialog.

Select a target processor (core)

Processor options affect the invocation of all tools in the toolset. In Eclipse you only need to set them once. Based on the target processor, the compiler includes a special function register file. This is a regular include file which enables you to use virtual registers that are located in memory.

1. From the **Project** menu, select **Properties for**

   The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Processor**.

   In the right pane the Processor page appears.

3. From the **Processor selection** list, select a processor.

To access the C/C++ compiler options

1. From the **Project** menu, select **Properties for**

   The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler**.

4.   Select the sub-entries and set the options in the various pages.

> Note that C++ compiler options are only enabled if you have added a C++ file to your project, a
> file with the extension `.cc`, `.cpp` or `.cxx`.

> Note that the options you enter in the Assembler page are also used for intermediate assembly
> files.

You can find a detailed description of all C++ compiler options in Section 10.3, *C++ Compiler Options*.

Invocation syntax on the command line (Windows Command Prompt):

cptc [[*option*]... [*file*]...]...

5.2. How the C++ Compiler Searches Include Files

When you use include files (with the `#include` statement), you can specify their location in several ways.
The C++ compiler searches the specified locations in the following order:

1. If the `#include` statement contains an absolute pathname, the C++ compiler looks for this file. If no
   path or a relative path is specified, the C++ compiler looks in the same directory as the source file.
   This is only possible for include files that are enclosed in "".

> This first step is not done for include files enclosed in <>.

2. When the C++ compiler did not find the include file, it looks in the directories that are specified in the
   C/C++ Compiler » Include Paths page in the **C/C++ Build » Settings » Tool Settings** tab of the
   Project Properties dialog (equivalent to the **--include-directory** (**-I**) command line option).

3. When the C++ compiler did not find the include file (because it is not in the specified include directory
   or because no directory is specified), it looks in the path(s) specified in the environment variable
   `CPTCINC`.

4. When the C++ compiler still did not find the include file, it finally tries the default `include.cpp` and
   `include` directory relative to the installation directory.

5. If the include file is still not found, the directories specified in the **--sys-include** option are searched.

If the include directory is specified as **-**, e.g., **-I-**, the option indicates the point in the list of **-I** or
--include-directory options at which the search for file names enclosed in < . . . > should begin. That is,
the search for < . . . > names should only consider directories named in **-I** or **--include-directory** options
following the **-I-**, and the directories of items 3 and 4 above. **-I-** also removes the directory containing the
current input file (item 1 above) from the search path for file names enclosed in " . . . ".

An include directory specified with the **--sys-include** option is considered a "system" include directory.
Warnings are suppressed when processing files found in system include directories.

If the filename has no suffix it will be searched for by appending each of a set of include file suffixes. When searching in a given directory all of the suffixes are tried in that directory before moving on to the next search directory. The default set of suffixes is, no extension and `.stdh`. The default can be overridden using the **--incl-suffixes** command line option. A null file suffix cannot be used unless it is present in the suffix list (that is, the C++ compiler will always attempt to add a suffix from the suffix list when the filename has no suffix).

Example

Suppose that the C++ source file `test.cc` contains the following lines:

```
#include <stdio.h>
#include "myinc.h"
```

You can call the C++ compiler as follows:

```
cptc -Imyinclude test.cc
```

First the C++ compiler looks for the file `stdio.h` in the directory `myinclude` relative to the current directory. If it was not found, the C++ compiler searches in the environment variable `CPTCINC` and then in the default `include` directory.

The C++ compiler now looks for the file `myinc.h`, in the directory where `test.cc` is located. If the file is not there the C++ compiler searches in the directory `myinclude`. If it was still not found, the C++ compiler searches in the environment variable `CPTCINC` and then in the default `include.cpp` and `include` directories.

5.3. C++ Compiler Error Messages

The C++ compiler reports the following types of error messages in the Problems view of Eclipse.

F (Fatal errors)

Catastrophic errors, also called 'fatal errors', indicate problems of such severity that the compilation cannot continue. For example: command-line errors, internal errors, and missing include files. If multiple source files are being compiled, any source files after the current one will not be compiled.

E (Errors)

Errors indicate violations of the syntax or semantic rules of the C++ language. Compilation continues, but object code is not generated.

W (Warnings)

Warnings indicate something valid but questionable. Compilation continues and object code is generated (if no errors are detected). You can control warnings in the **C/C++ Build » Settings » Tool Settings » C/C++ Compiler » Diagnostics** page of the **Project » Properties for** menu (C++ compiler option **--no-warnings**).

R (Remarks)

Remarks indicate something that is valid and probably intended, but which a careful programmer may want to check. These diagnostics are not issued by default. Compilation continues and object code is generated (if no errors are detected). To enable remarks, enable the option **Issue remarks on C++ code** in the **C/C++ Build » Settings » Tool Settings » C/C++ Compiler » Diagnostics** page of the **Project » Properties for** menu (C++ compiler option **--remarks**).

S (Internal errors)

Internal compiler errors are caused by failed internal consistency checks and should never occur. However, if such a 'SYSTEM' error appears, please report the occurrence to Altium. Please include a small C++ program causing the error.

Message format

By default, diagnostics are written in a form like the following:

```
cptc E0020: ["test.cc" 3] identifier "name" is undefined
```

With the command line option **--error-file=***file* you can redirect messages to a file instead of `stderr`.

Note that the message identifies the file and line involved. Long messages are wrapped to additional lines when necessary.

With the option **C/C++ Build » Settings » Tool Settings » Global Options » Treat warnings as errors** (option **--warnings-as-errors**) you can change the severity of warning messages to errors.

With the command line option **--diag** you can see a list of all messages.

For some messages, a list of entities is useful; they are listed following the initial error message:

```
cptc E0308: ["test.cc" 4] more than one instance of overloaded
        function "f" matches the argument list:
            function "f(int)"
            function "f(float)"
            argument types are: (double)
```

In some cases, some additional context information is provided; specifically, such context information is useful when the C++ compiler issues a diagnostic while doing a template instantiation or while generating a constructor, destructor, or assignment operator function. For example:

```
cptc E0265: ["test.cc" 7] "A::A()" is inaccessible
        detected during implicit generation of "B::B()" at line 7
```

Without the context information, it is very hard to figure out what the error refers to.

Termination Messages

The C++ compiler writes sign-off messages to `stderr` (the Problems view in Eclipse) if errors are detected. For example, one of the following forms of message

```
n errors detected in the compilation of "file".
```

```
1 catastrophic error detected in the compilation of "file".
```

```
n errors and 1 catastrophic error detected in the compilation of "file".
```

is written to indicate the detection of errors in the compilation. No message is written if no errors were detected. The following message

```
Error limit reached.
```

is written when the count of errors reaches the error limit (see the option **--error-limit**); compilation is then terminated. The message

```
Compilation terminated.
```

is written at the end of a compilation that was prematurely terminated because of a catastrophic error. The message

```
Compilation aborted
```

is written at the end of a compilation that was prematurely terminated because of an internal error. Such an error indicates an internal problem in the compiler. If such an internal error appears, please report the occurrence to Altium. Please include a small C++ program causing the error.

Chapter 6. Using the Assembler

This chapter describes the assembly process and explains how to call the assembler.

The assembler converts hand-written or compiler-generated assembly language programs into machine language, resulting in object files in the ELF/DWARF object format.

The assembler takes the following files for input and output:
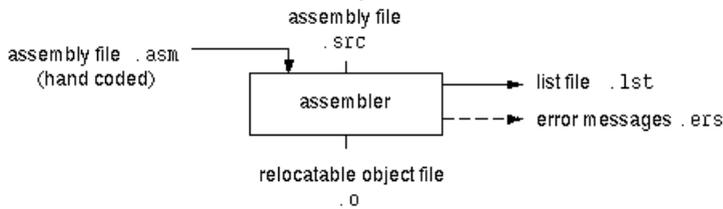
The following information is described:

• The assembly process.

• How to call the assembler and how to use its options. An extensive list of all options and their descriptions is included in Section 10.4, *Assembler Options*.

• The various assembler optimizations.

• How to generate a list file.

• Types of assembler messages.

6.1. Assembly Process

The assembler generates relocatable output files with the extension `.o`. These files serve as input for the linker.

Phases of the assembly process

• Parsing of the source file: preprocessing of assembler directives and checking of the syntax of instructions

• Instruction grouping and reordering

• Optimization (instruction size and generic instructions)

• Generation of the relocatable object file and optionally a list file

The assembler integrates file inclusion and macro facilities. See Section 3.10, *Macro Operations* for more information.

6.2. Calling the Assembler

The TASKING VX-toolset for TriCore under Eclipse can use the internal builder (default) or the TASKING makefile generator (external builder) to build your entire project. After you have built your project, the output files are available in a subdirectory of your project directory, depending on the active configuration you have set in the **C/C++ Build » Settings** page of the **Project » Properties for** dialog.

Building a project under Eclipse

You have several ways of building your project:

- Build Selected File(s) (▣). This compiles and assembles the selected file(s) without calling the linker.

    1. In the C/C++ Projects view, select the files you want to compile.

    2. Right-click in the C/C++ Projects view and select **Build Selected File(s)**.

- Build Individual Project (▣).

    To build individual projects incrementally, select **Project » Build** *project*.

- Rebuild Project (▣). This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.

    1. Select **Project » Clean...**

    2. Enable the option **Start a build immediately** and click **OK**.

- Build Automatically. This performs a build of all projects whenever any project file is saved, such as your makefile.

    This way of building is not recommended for C/C++ development, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item. In order for this option to work, you must also enable option **Build on resource save (Auto build)** on the **Behaviour** tab of the **C/C++ Build** page of the **Project » Properties for** dialog.

Select a target processor (core)

Processor options affect the invocation of all tools in the toolset. In Eclipse you only need to set them once. Based on the target processor, the compiler includes a special function register file. This is a regular include file which enables you to use virtual registers that are located in memory.

You can specify the target processor when you create a new project with the New C/C++ Project wizard (**File » New » TASKING TriCore C/C++ Project**), but you can always change the processor in the project properties dialog.

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Processor**.

In the right pane the Processor page appears.

3. From the **Processor selection** list, select a processor.

To access the assembler options

1. From the **Project** menu, select **Properties for**

   The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   In the right pane the Settings appear.

3. On the Tool Settings tab, select **Assembler**.

4. Select the sub-entries and set the options in the various pages.

> Note that the options you enter in the Assembler page are not only used for hand-coded assembly files, but also for the assembly files generated by the compiler.

You can find a detailed description of all assembler options in Section 10.4, *Assembler Options*.

Invocation syntax on the command line (Windows Command Prompt):

astc [[*option*]... [*file*]...]...

The input file must be an assembly source file (.asm or .src).

6.3. How the Assembler Searches Include Files

When you use include files (with the .INCLUDE directive), you can specify their location in several ways. The assembler searches the specified locations in the following order:

1. If the .INCLUDE directive contains an absolute path name, the assembler looks for this file. If no path or a relative path is specified, the assembler looks in the same directory as the source file.

2. When the assembler did not find the include file, it looks in the directories that are specified in the **Assembler » Include Paths** page in the **C/C++ Build » Settings » Tool Settings** tab of the Project Properties dialog (equivalent to the **-I** command line option).

3. When the assembler did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks in the path(s) specified in the environment variable ASTCINC.

4. When the assembler still did not find the include file, it finally tries the default include directory relative to the installation directory.

Example

Suppose that the assembly source file `test.asm` contains the following lines:

```
.INCLUDE 'myinc.inc'
```

You can call the assembler as follows:

```
astc -Imyinclude test.asm
```

First the assembler looks for the file `myinc.asm`, in the directory where `test.asm` is located. If the file is not there the assembler searches in the directory `myinclude`. If it was still not found, the assembler searches in the environment variable `ASTCINC` and then in the default `include` directory.

6.4. Assembler Optimizations

The assembler can perform various optimizations that you can enable or disable.

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **Assembler » Optimization**.

4. Enable one or more optimizations.

You can enable or disable the optimizations described below. The command line option for each optimization is given in brackets.

Allow generic instructions (option -Og/-OG)

When this option is enabled, you can use generic instructions in your assembly source. The assembler tries to replace instructions by faster or smaller instructions. For example, the instruction `jeq d0,#0,label1` is replaced by `jz d0,label1`.

By default this option is enabled. Because shorter instructions may influence the number of cycles, you may want to disable this option when you have written timed code. In that case the assembler encodes all instructions as they are.

Optimize instruction size (option -Os/-OS)

When this option is enabled, the assembler tries to find the shortest possible operand encoding for instructions. By default this option is enabled.

6.5. Generating a List File

The list file is an additional output file that contains information about the generated code. You can customize the amount and form of information.

If the assembler generates errors or warnings, these are reported in the list file just below the source line that caused the error or warning.

To generate a list file

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **Assembler » List File**.

4. Enable the option **Generate list file**.

5. (Optional) Enable the options to include that information in the list file.

Example on the command line (Windows Command Prompt)

The following command generates the list file `test.lst`:

```
astc -l test.asm
```

See Section 14.1, *Assembler List File Format*, for an explanation of the format of the list file.

6.6. Assembler Error Messages

The assembler reports the following types of error messages in the Problems view of Eclipse.

F (Fatal errors)

After a fatal error the assembler immediately aborts the assembly process.

E (Errors)

Errors are reported, but the assembler continues assembling. No output files are produced unless you have set the assembler option **--keep-output-files** (the resulting output file may be incomplete).

W (Warnings)

Warning messages do not result into an erroneous assembly output file. They are meant to draw your attention to assumptions of the assembler for a situation which may not be correct. You can control

warnings in the **C/C++ Build » Settings » Tool Settings » Assembler » Diagnostics** page of the **Project » Properties for** menu (assembler option **--no-warnings**).

Display detailed information on diagnostics

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

    The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

    A popup menu appears.

3. Select **Detailed Diagnostics Info**.

    A dialog box appears with additional information.

On the command line you can use the assembler option **--diag** to see an explanation of a diagnostic message:

```
astc --diag=[format:]{all | number,...]
```

Chapter 7. Using the Linker

This chapter describes the linking process, how to call the linker and how to control the linker with a script file.

The TASKING linker is a combined linker/locator. The linker phase combines relocatable object files (`.o` files, generated by the assembler), and libraries into a single relocatable linker object file (`.out`). The locator phase assigns absolute addresses to the linker object file and creates an absolute object file which you can load into a target processor. From this point the term linker is used for the combined linker/locator.

The linker can simultaneously link and locate all programs for all cores available on a target board. The target board may be of arbitrary complexity. A simple target board may contain one standard processor with some external memory that executes one task. A complex target board may contain multiple standard processors and DSPs combined with configurable IP-cores loaded in an FPGA. Each core may execute a different program, and external memory may be shared by multiple cores.

The linker takes the following files for input and output:
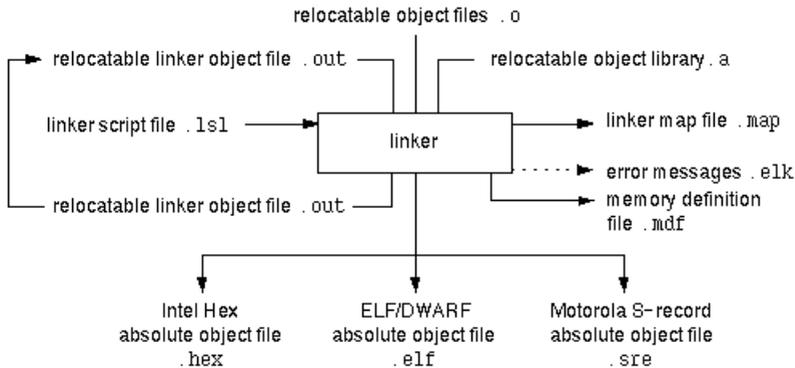
This chapter first describes the linking process. Then it describes how to call the linker and how to use its options. An extensive list of all options and their descriptions is included in Section 10.5, *Linker Options*.

To control the link process, you can write a script for the linker. This chapter shortly describes the purpose and basic principles of the Linker Script Language (LSL) on the basis of an example. A complete description of the LSL is included in Linker Script Language.

7.1. Linking Process

The linker combines and transforms relocatable object files (`.o`) into a single absolute object file. This process consists of two phases: the linking phase and the locating phase.

In the first phase the linker combines the supplied relocatable object files and libraries into a single relocatable object file. In the second phase, the linker assigns absolute addresses to the object file so it can actually be loaded into a target.

Terms used in the linking process

Term	Definition
Absolute object file	Object code in which addresses have fixed absolute values, ready to load into a target.
Address	A specification of a location in an address space.
Address space	The set of possible addresses. A core can support multiple spaces, for example in a Harvard architecture the addresses that identify the location of an instruction refer to *code* space, whereas addresses that identify the location of a data object refer to a *data* space.
Architecture	A description of the characteristics of a core that are of interest for the linker. This encompasses the address space(s) and the internal bus structure. Given this information the linker can convert logical addresses into physical addresses.
Copy table	A section created by the linker. This section contains data that specifies how the startup code initializes the data and BSS sections. For each section the copy table contains the following fields:
• action: defines whether a section is copied or zeroed
• destination: defines the section's address in RAM
• source: defines the sections address in ROM, zero for BSS sections
• length: defines the size of the section in MAUs of the destination space
Core	An instance of an architecture.
Derivative	The design of a processor. A description of one or more cores including internal memory and any number of buses.
Library	Collection of relocatable object files. Usually each object file in a library contains one symbol definition (for example, a function).
Link task	A scope for linking: resolving symbols from object files and libraries. Such a task is associated with one core in the LSL file(s). Other LSL cores may be imported into this core, associating two or more hardware cores with one link task.
Logical address	An address as encoded in an instruction word, an address generated by a core (CPU).
LSL file	The set of linker script files that are passed to the linker.
MAU	Minimum Addressable Unit. For a given processor the number of bits between an address and the next address. This is not necessarily a byte or a word.
Object code	The binary machine language representation of the C source.
Physical address	An address generated by the memory system.
Processor	An instance of a derivative. Usually implemented as a (custom) chip, but can also be implemented in an FPGA, in which case the derivative can be designed by the developer.

Term	Definition
Relocatable object file	Object code in which addresses are represented by symbols and thus relocatable.
Relocation	The process of assigning absolute addresses.
Relocation information	Information about how the linker must modify the machine code instructions when it relocates addresses.
Section	A group of instructions and/or data objects that occupy a contiguous range of addresses.
Section attributes	Attributes that define how the section should be linked or located.
Target	The hardware board on which an application is executing. A board contains at least one processor. However, a complex target may contain multiple processors and external memory and may be shared between processors.
Unresolved reference	A reference to a symbol for which the linker did not find a definition yet.

7.1.1. Phase 1: Linking

The linker takes one or more relocatable object files and/or libraries as input. A relocatable object file, as generated by the assembler, contains the following information:

- *Header information*: Overall information about the file, such as the code size, name of the source file it was assembled from, and creation date.

- *Object code*: Binary code and data, divided into various named sections. Sections are contiguous chunks of code that have to be placed in specific parts of the memory. The program addresses start at zero for each section in the object file.

- *Symbols*: Some symbols are exported - defined within the file for use in other files. Other symbols are imported - used in the file but not defined (external symbols). Generally these symbols are names of routines or names of data objects.

- *Relocation information*: A list of places with symbolic references that the linker has to replace with actual addresses. When in the code an external symbol (a symbol defined in another file or in a library) is referenced, the assembler does not know the symbol's size and address. Instead, the assembler generates a call to a preliminary relocatable address (usually 0000), while stating the symbol name.

- *Debug information*: Other information about the object code that is used by a debugger. The assembler optionally generates this information and can consist of line numbers, C source code, local symbols and descriptions of data structures.

The linker resolves the external references between the supplied relocatable object files and/or libraries and combines the files into a single relocatable linker object file.

The linker starts its task by scanning all specified relocatable object files and libraries. If the linker encounters an unresolved symbol, it remembers its name and continues scanning. The symbol may be defined elsewhere in the same file, or in one of the other files or libraries that you specified to the linker. If the symbol is defined in a library, the linker extracts the object file with the symbol definition from the library. This way the linker collects all definitions and references of all of the symbols.

Next, the linker combines sections with the same section name and attributes into single sections. The linker also substitutes (external) symbol references by (relocatable) numerical addresses where possible. At the end of the linking phase, the linker either writes the results to a file (a single relocatable object file) or keeps the results in memory for further processing during the locating phase.

The resulting file of the linking phase is a single relocatable object file (.out). If this file contains unresolved references, you can link this file with other relocatable object files (.o) or libraries (.a) to resolve the remaining unresolved references.

With the linker command line option **--link-only**, you can tell the linker to only perform this linking phase and skip the locating phase. The linker complains if any unresolved references are left.

7.1.2. Phase 2: Locating

In the locating phase, the linker assigns absolute addresses to the object code, placing each section in a specific part of the target memory. The linker also replaces references to symbols by the actual address of those symbols. The resulting file is an absolute object file which you can actually load into a target memory. Optionally, when the resulting file should be loaded into a ROM device the linker creates a so-called copy table section which is used by the startup code to initialize the data and BSS sections.

Code modification

When the linker assigns absolute addresses to the object code, it needs to modify this code according to certain rules or *relocation expressions* to reflect the new addresses. These relocation expressions are stored in the relocatable object file. Consider the following snippet of x86 code that moves the contents of variable a to variable b via the eax register:

```
A1 3412 0000 mov a,%eax    (a defined at 0x1234, byte reversed)
A3 0000 0000 mov %eax,b    (b is imported so the instruction refers to
                             0x0000 since its location is unknown)
```

Now assume that the linker links this code so that the section in which a is located is relocated by 0x10000 bytes, and b turns out to be at 0x9A12. The linker modifies the code to be:

```
A1 3412 0100 mov a,%eax    (0x10000 added to the address)
A3 129A 0000 mov %eax,b    (0x9A12 patched in for b)
```

These adjustments affect instructions, but keep in mind that any pointers in the data part of a relocatable object file have to be modified as well.

Output formats

The linker can produce its output in different file formats. The default ELF/DWARF format (.elf) contains an image of the executable code and data, and can contain additional debug information. The Intel-Hex format (.hex) and Motorola S-record format (.sre) only contain an image of the executable code and data. You can specify a format with the options **--output** (**-o**) and **--chip-output** (**-c**).

Controlling the linker

Via a so-called *linker script file* you can gain complete control over the linker. The script language is called the *Linker Script Language* (LSL). Using LSL you can define:

- The memory installed in the embedded target system:

  To assign locations to code and data sections, the linker must know what memory devices are actually installed in the embedded target system. For each physical memory device the linker must know its start-address, its size, and whether the memory is read-write accessible (RAM) or read-only accessible (ROM).

- How and where code and data should be placed in the physical memory:

  Embedded systems can have complex memory systems. If for example on-chip and off-chip memory devices are available, the code and data located in internal memory is typically accessed faster and with dissipating less power. To improve the performance of an application, specific code and data sections should be located in on-chip memory. By writing your own LSL file, you gain full control over the locating process.

- The underlying hardware architecture of the target processor.

  To perform its task the linker must have a model of the underlying hardware architecture of the processor you are using. For example the linker must know how to translate an address used within the object file (a logical address) into an offset in a particular memory device (a physical address). In most linkers this model is hard coded in the executable and can not be modified. For the TASKING linker this hardware model is described in the linker script file. This solution is chosen to support configurable cores that are used in system-on-chip designs.

When you want to write your own linker script file, you can use the standard linker script files with architecture descriptions delivered with the product.

See also Section 7.8, *Controlling the Linker with a Script*.

7.2. Calling the Linker

In Eclipse you can set options specific for the linker. After you have built your project, the output files are available in a subdirectory of your project directory, depending on the active configuration you have set in the **C/C++ Build » Settings** page of the **Project » Properties for** dialog.

Building a project under Eclipse

You have several ways of building your project:

- Build Individual Project (⬚).

  To build individual projects incrementally, select **Project » Build** *project*.

- Rebuild Project (⬚). This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.

  1. Select **Project » Clean...**

  2. Enable the option **Start a build immediately** and click **OK**.

- Build Automatically. This performs a build of all projects whenever any project file is saved, such as your makefile.

  This way of building is not recommended, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item. In order for this option to work, you must also enable option **Build on resource save (Auto build)** on the **Behaviour** tab of the **C/C++ Build** page of the **Project » Properties for** dialog.

To access the linker options

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker**.

4. Select the sub-entries and set the options in the various pages.

You can find a detailed description of all linker options in Section 10.5, *Linker Options*.

Invocation syntax on the command line (Windows Command Prompt):

ltc [[*option*]... [*file*]...]...

When you are linking multiple files, either relocatable object files (.o) or libraries (.a), it is important to specify the files in the right order. This is explained in Section 7.3, *Linking with Libraries*.

Example:

```
ltc -dtc1796b.lsl test.o
```

This links and locates the file test.o and generates the file test.elf.

7.3. Linking with Libraries

There are two kinds of libraries: system libraries and user libraries.

System library

System libraries are stored in the directories:

```
<TriCore installation path>\lib\tc13      (TriCore 1.3 libraries)
<TriCore installation path>\lib\tc13_mmu  (MMU variant)
<TriCore installation path>\lib\tc131     (TriCore 1.3.1 libraries)
<TriCore installation path>\lib\tc16      (TriCore 1.6 libraries)
<TriCore installation path>\lib\tc16_mmu  (MMU variant)
```

```
<TriCore installation path>\lib\tc16x       (TriCore 1.6.x libraries)
<TriCore installation path>\lib\tc162       (TriCore 1.6.2 libraries)
<TriCore installation path>\lib\p           (protected libraries)
```

The p directory contains subdirectories with the protected libraries for CPU functional problems.

An overview of the system libraries is given in the following table:

Libraries	Description
libc[s].a libc[s]_fpu.a	C libraries Optional letter: s = single precision floating-point (control program option **--fp-model=+float**) _fpu = with FPU instructions (default, control program option **--fp-model=-soft**)
libfp[t].a libfp[t]_fpu.a	Floating-point libraries Optional letter: t = trapping (control program option **--fp-model=+trap**) _fpu = with FPU instructions (default, control program option **--fp-model=-soft**)
librt.a	Run-time library
libpb.a libpc.a libpct.a libpd.a libpt.a	Profiling libraries pb = block/function counter pc = call graph pct = call graph and timing pd = dummy pt = function timing
libcp[s][x].a libcp[s][x]_fpu.a	C++ libraries Optional letter: s = single precision floating-point x = exception handling _fpu = with FPU instructions (default, control program option **--fp-model=-soft**)
libstl[s]x.a libstl[s]x_fpu.a	STLport C++ libraries (exception handling variants only) Optional letter: s = single precision floating-point _fpu = with FPU instructions (default, control program option **--fp-model=-soft**)

Sources for the libraries are present in the directories lib\src, lib\src.* in the form of an executable. If you run the executable it will extract the sources in the corresponding directory.

To link the default C (system) libraries

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2.   In the left pane, expand **C/C++ Build** and select **Settings**.

    In the right pane the Settings appear.

3.   On the Tool Settings tab, select **Linker » Libraries**.

4.   Enable the option **Link default libraries**.

When you want to link system libraries from the command line, you must specify this with the option **--library** (**-l**). For example, to specify the system library `libc.a`, type:

```
ltc --library=c test.o
```

User library

You can create your own libraries. Section 8.5, *Archiver* describes how you can use the archiver to create your own library with object modules.

To link user libraries

1.   From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2.   In the left pane, expand **C/C++ Build** and select **Settings**.

    In the right pane the Settings appear.

3.   On the Tool Settings tab, select **Linker » Libraries**.

4.   Add your libraries to the **Libraries** box.

When you want to link user libraries from the command line, you must specify their filenames on the command line:

```
ltc start.o mylib.a
```

If the library resides in a sub-directory, specify that directory with the library name:

```
ltc start.o mylibs\mylib.a
```

If you do not specify a directory, the linker searches the library in the current directory only.

Library order

The order in which libraries appear on the command line is important. By default the linker processes object files and libraries in the order in which they appear at the command line. Therefore, when you use a weak symbol construction, like `printf`, in an object file or your own library, you must position this object/library before the C library.

With the option **--first-library-first** you can tell the linker to scan the libraries from left to right, and extract symbols from the first library where the linker finds it. This can be useful when you want to use newer versions of a library routine:

```
ltc --first-library-first a.a test.o b.a
```

If the file `test.o` calls a function which is both present in `a.a` and `b.a`, normally the function in `b.a` would be extracted. With this option the linker first tries to extract the symbol from the first library `a.a`.

Note that routines in `b.a` that call other routines that are present in both `a.a` and `b.a` are now also resolved from `a.a`.

7.3.1. How the Linker Searches Libraries

System libraries

You can specify the location of system libraries in several ways. The linker searches the specified locations in the following order:

1. The linker first looks in the **Library search path** that are specified in the **Linker » Libraries** page in the **C/C++ Build » Settings » Tool Settings** tab of the Project Properties dialog (equivalent to the **-L** command line option). If you specify the **-L** option without a pathname, the linker stops searching after this step.

2. When the linker did not find the library (because it is not in the specified library directory or because no directory is specified), it looks in the path(s) specified in the environment variables `LIBTC1V1_3` / `LIBTC1V1_3_1` / `LIBTC1V1_6` / `LIBTC1V1_6_X` / `LIBTC1V1_6_2`.

3. When the linker did not find the library, it tries the default `lib` directory relative to the installation directory (or a processor specific sub-directory).

User library

If you use your own library, the linker searches the library in the current directory only.

7.3.2. How the Linker Extracts Objects from Libraries

A library built with the TASKING archiver **artc** always contains an index part at the beginning of the library. The linker scans this index while searching for unresolved externals. However, to keep the index as small as possible, only the defined symbols of the library members are recorded in this area.

When the linker finds a symbol that matches an unresolved external, the corresponding object file is extracted from the library and is processed. After processing the object file, the remaining library index is searched. If after a complete search of the library unresolved externals are introduced, the library index will be scanned again. After all files and libraries are processed, and there are still unresolved externals and you did not specify the linker option **--no-rescan**, all libraries are rescanned again. This way you do not have to worry about the library order on the command line and the order of the object files in the libraries. However, this rescanning does not work for 'weak symbols'. If you use a weak symbol construction, like `printf`, in an object file or your own library, you must position this object/library before the C library.

The option **--verbose** (**-v**) shows how libraries have been searched and which objects have been extracted.

Resolving symbols

If you are linking from libraries, only the objects that contain symbols to which you refer, are extracted from the library. This implies that if you invoke the linker like:

```
ltc mylib.a
```

nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through `mylib.a`.

It is possible to force a symbol as external (unresolved symbol) with the option **--extern** (**-e**):

```
ltc --extern=main mylib.a
```

In this case the linker searches for the symbol `main` in the library and (if found) extracts the object that contains `main`.

If this module contains new unresolved symbols, the linker looks again in `mylib.a`. This process repeats until no new unresolved symbols are found.

7.4. Incremental Linking

With the TASKING linker it is possible to link incrementally. Incremental linking means that you link some, but not all `.o` modules to a relocatable object file `.out`. In this case the linker does not perform the locating phase. With the second invocation, you specify both new `.o` files as the `.out` file you had created with the first invocation.

> Incremental linking is only possible on the command line.

```
ltc --incremental test1.o -otest.out
ltc test2.o test.out
```

This links the file `test1.o` and generates the file `test.out`. This file is used again and linked together with `test2.o` to create the file `test.elf` (the default name if no output filename is given in the default ELF/DWARF format).

With incremental linking it is normal to have unresolved references in the output file until all `.o` files are linked and the final `.out` or `.elf` file has been reached. The option **--incremental** (**-r**) for incremental linking also suppresses warnings and errors because of unresolved symbols.

7.5. Linking For a Multi-Core Processor in a Single-Core Configuration

The TASKING VX-toolset for TriCore has support for multi-core versions of the TriCore. Multi-core is supported for the TriCore 1.6.x architecture only. For example, the AURIX family TC27x contains three TriCore cores (core 0, core 1 and core 2). To build an application for such a multi-core processor it is sufficient to specify the correct processor in Eclipse or to the control program (control program option

--cpu=tc27x). If you want to build an application for a single-core configuration, you also need to select the specific core in Eclipse (for example, TriCore core 0). If you build your sources on the command line with the control program, you also have to specify control program option **--lsl-core=tc0**.

To select a single-core configuration

1. From the **Project** menu, select **Properties for**

   The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Processor**.

   In the right pane the Processor settings appear.

3. From the **Processor Selection** list, select a processor or select **User defined TriCore ...**.

4. From the **Multi-core configuration** list, select a TriCore single-core.

When you build your project, the linker is called with linker option **--core=mpe:tc0** and the correct LSL file is used.

7.6. Importing Binary Files

With the TASKING linker it is possible to add a binary file to your absolute output file. In an embedded application you usually do not have a file system where you can get your data from. With the linker option **--import-object** you can add raw data to your application. This makes it possible for example to display images on a device or play audio. The linker puts the raw data from the binary file in a section. The section is aligned on a 4-byte boundary. The section name is derived from the filename, in which dots are replaced by an underscore. So, when importing a file called `my.mp3`, a section with the name `my_mp3` is created. In your application you can refer to the created section by using linker labels.

For example:

```
#include <stdio.h>
__far extern char   _lc_ub_my_mp3; /* linker labels */
__far extern char   _lc_ue_my_mp3;
char*   mp3 = &_lc_ub_my_mp3;

void main(void)
{
  int size = &_lc_ue_my_mp3 - &_lc_ub_my_mp3;
  int i;
  for (i=0;i<size;i++)
    putchar(mp3[i]);
}
```

> Because the compiler does not know in which space the linker will locate the imported binary, you have to make sure the symbols refer to the same space in which the linker will place the imported binary. You do this by using the memory qualifier `__far`, otherwise the linker cannot bind your linker symbols.
>
> Also note that if you want to use the export functionality of Eclipse, the binary file has to be part of your project.

7.7. Linker Optimizations

During the linking and locating phase, the linker looks for opportunities to optimize the object code. Both code size and execution speed can be optimized.

To enable or disable optimizations

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Optimization**.

4. Enable one or more optimizations.

You can enable or disable the optimizations described below. The command line option for each optimization is given in brackets.

Delete unreferenced sections (option -Oc/-OC)

This optimization removes unused sections from the resulting object file.

First fit decreasing (option -Ol/-OL)

When the physical memory is fragmented or when address spaces are nested it may be possible that a given application cannot be located although the size of the available physical memory is larger than the sum of the section sizes. Enable the first-fit-decreasing optimization when this occurs and re-link your application.

The linker's default behavior is to place sections in the order that is specified in the LSL file (that is, working from low to high memory addresses or vice versa). This also applies to sections within an unrestricted group. If a memory range is partially filled and a section must be located that is larger than the remainder of this range, then the section and all subsequent sections are placed in a next memory range. As a result of this gaps occur at the end of a memory range.

When the first-fit-decreasing optimization is enabled the linker will first place the largest sections in the smallest memory ranges that can contain the section. Small sections are located last and can likely fit in the remaining gaps.

Compress copy table (option -Ot/-OT)

The startup code initializes the application's data areas. The information about which memory addresses should be zeroed and which memory ranges should be copied from ROM to RAM is stored in the copy table.

When this optimization is enabled the linker will try to locate sections in such a way that the copy table is as small as possible thereby reducing the application's ROM image.

Delete duplicate code (option -Ox/-OX)

Delete duplicate constant data (option -Oy/-OY)

These two optimizations remove code and constant data that is defined more than once, from the resulting object file.

7.8. Controlling the Linker with a Script

With the options on the command line you can control the linker's behavior to a certain degree. From Eclipse it is also possible to determine where your sections will be located, how much memory is available, which sorts of memory are available, and so on. Eclipse passes these locating directions to the linker via a script file. If you want even more control over the locating process you can supply your own script.

The language for the script is called the *Linker Script Language*, or shortly LSL. You can specify the script file to the linker, which reads it and locates your application exactly as defined in the script. If you do not specify your own script file, the linker always reads a standard script file which is supplied with the toolset.

7.8.1. Purpose of the Linker Script Language

The Linker Script Language (LSL) serves three purposes:

1. It provides the linker with a definition of the target's core architecture. This definition is supplied with the toolset.

2. It provides the linker with a specification of the memory attached to the target processor.

3. It provides the linker with information on how your application should be located in memory. This gives you, for example, the possibility to create overlaying sections.

The linker accepts multiple LSL files. You can use the specifications of the core architectures that Altium has supplied in the `include.lsl` directory. Do not change these files.

If you use a different memory layout than described in the LSL file supplied for the target core, you must specify this in a separate LSL file and pass both the LSL file that describes the core architecture and your

LSL file that contains the memory specification to the linker. Next you may want to specify how sections should be located and overlaid. You can do this in the same file or in another LSL file.

LSL has its own syntax. In addition, you can use the standard C preprocessor keywords, such as `#include` and `#define`, because the linker sends the script file first to the C preprocessor before it starts interpreting the script.

The complete LSL syntax is described in Chapter 16, *Linker Script Language (LSL)*.

7.8.2. Eclipse and LSL

In Eclipse you can specify the size of the stack and heap; the physical memory attached to the processor; identify that particular address ranges are reserved; and specify which sections are located where in memory. Eclipse translates your input into an LSL file that is stored in the project directory under the name *project_name*.`lsl` and passes this file to the linker. If you want to learn more about LSL you can inspect the generated file *project_name*.`lsl`.

To add a generated Linker Script File to your project

1. From the **File** menu, select **File » New » TASKING TriCore C/C++ Project**.

    The New C/C++ Project wizard appears.

2. Fill in the project settings in each dialog and click **Next >** until the following dialog appears.
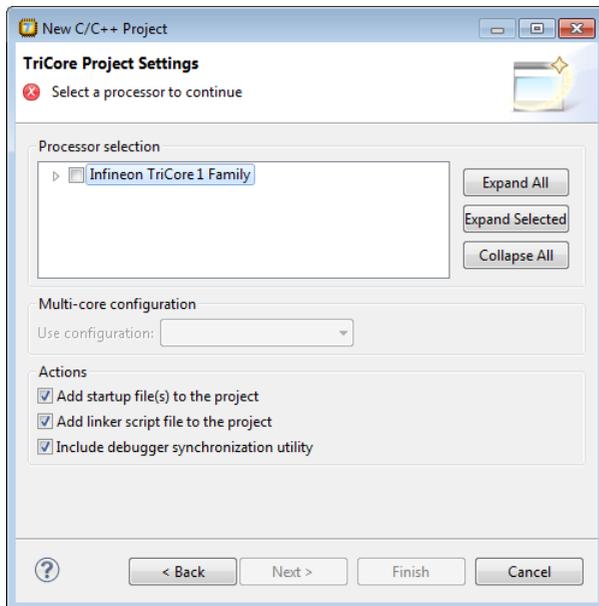
3. Enable the option **Add linker script file to the project** and click **Finish**.

    Eclipse creates your project and the file "project_name.lsl" in the project directory.

If you do not add the linker script file here, you can always add it later with **File » New » Linker Script File (LSL)**.

To change the Linker Script File in Eclipse

There are two ways of changing the LSL file in Eclipse.

- You can change the LSL file directly in an editor.

    1. Double-click on the file *project_name*.lsl.

        The project LSL file opens in the editor area.
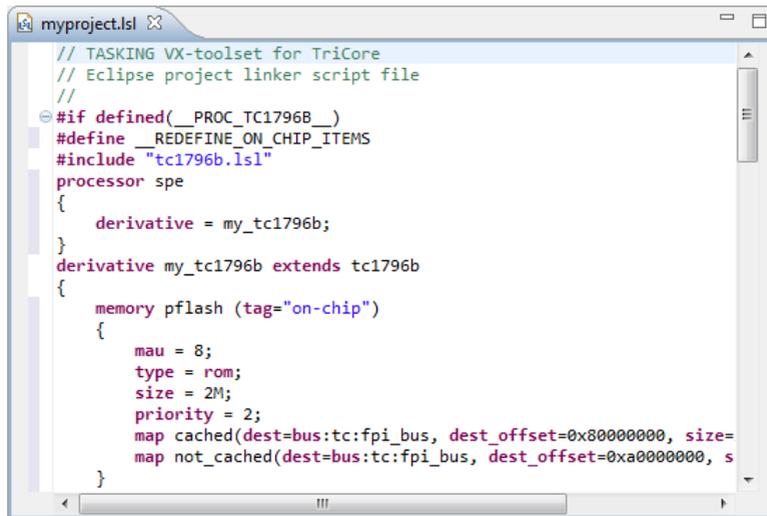
    2. You can edit the LSL file directly in the *project_name*.lsl editor.

        A * appears in front of the name of the LSL file to indicate that the file has changes.

    3. Click 🖫 or select **File » Save** to save the changes.

- You can also make changes to the property pages Memory and Stack/Heap.

    1. From the **Project** menu, select **Properties for**

        The Properties dialog appears.

    2. In the left pane, expand **C/C++ Build** and select **Memory** or **Stack/Heap**.

        In the right pane the corresponding property page appears.

    3. Make changes to memory and/or stack/heap and click **OK**.

        The project LSL file is updated automatically according to the changes you make in the pages.

You can quickly navigate through the LSL file by using the Outline view (**Window » Show View » Outline**).

7.8.3. Preprocessor Macros in the Linker Script Files

The linker script files contain several predefined preprocessor macros. If for some reason you need to change a default value, you can use Eclipse or the linker option **--define** to define a new value, or add this option via the control program to the linker.

For example, to set the user stack size from the command line to 24k, enter:

```
cctc -Wl--define=USTACK=24k test.c
```

With option **-Wl** the control program passes the macro definition to the linker.

The following macros are available in the linker script files.

Macro	Description
CSA	Specifies the size of the context save area (same as (CSA_TC0).
CSA_TC*n*	Specifies the size of the context save area for the specified TriCore core *n*.
CSA_START	Specifies the start address of the context save area (same as CSA_START_TC0).
CSA_START_TC*n*	Specifies the start address of the context save area for the specified TriCore core *n*.
HEAP	Specifies the size of the heap.
INTTAB	Specifies the start address of the interrupt table (same as INTTAB0).
INTTAB*n*	Specifies the start address of the interrupt table for the specified TriCore core *n*.
ISTACK	Specifies the size of the interrupt stack (same as ISTACK_TC0)..
ISTACK_TC*n*	Specifies the size of the interrupt stack for the specified TriCore core *n*.
MCS00_0RAM .. MCS*xx*_0RAM	Identifies the offset of the memory for core MCS00 .. MCS*xx* from the GTM base address.
RESET	Specifies the reset address.
TRAPTAB	Specifies the start address of the trap table (same as TRAPTAB0).
TRAPTAB*n*	Specifies the start address of the trap table for the specified TriCore core *n*.
USTACK	Specifies the size of the user stack (same as USTACK_TC0).
USTACK_TC*n*	Specifies the size of the user stack for the specified TriCore core *n*.
__REDEFINE_ON_CHIP_ITEMS	If defined as 1, no on-chip memories are defined.

7.8.4. Structure of a Linker Script File

A script file consists of several definitions. The definitions can appear in any order.

The architecture definition (required)

In essence an *architecture definition* describes how the linker should convert logical addresses into physical addresses for a given type of core. If the core supports multiple address spaces, then for each space the linker must know how to perform this conversion. In this context a physical address is an offset on a given internal or external bus. Additionally the architecture definition contains information about items such as the (hardware) stack and the interrupt vector table.

This specification is normally written by Altium. Altium supplies LSL files in the `include.lsl` directory. The file `tc_arch.lsl` defines the base architecture for all generic TriCore cores and includes an interrupt vector table (`inttab.lsl`) and an trap vector table (`traptab.lsl`). The file `tc_mc_arch.lsl` defines the base architecture for all multi-core TriCore cores. The files `tc1v1_3.lsl`, `tc1v1_3_1.lsl`, `tc1v1_6.lsl`, `tc1v1_6_x.lsl` and `tc1v1_6_2.lsl` extend the base architecture for each TriCore core.

The architecture definition of the LSL file should not be changed by you unless you also modify the core's hardware architecture. If the LSL file describes a multi-core system an architecture definition must be available for each different type of core.

The linker uses the architecture name in the LSL file to identify the target. For example, the default library search path can be different for each core architecture.

The derivative definition

The *derivative definition* describes the configuration of the internal (on-chip) bus and memory system. Basically it tells the linker how to convert offsets on the buses specified in the architecture definition into offsets in internal memory. Microcontrollers and DSPs often have internal memory and I/O sub-systems apart from one or more cores. The design of such a chip is called a *derivative*.

When you want to use multiple cores of the same type, you must instantiate the cores in a derivative definition, since the linker automatically instantiates only a single core for an unused architecture.

Altium supplies LSL files for each derivative (*derivative*`.lsl`), along with "SFR files", which provide easy access to registers in I/O sub-systems from C and assembly programs. When you build an ASIC or use a derivative that is not (yet) supported by the TASKING tools, you may have to write a derivative definition.

The processor definition

The *processor definition* describes an instance of a derivative. A processor definition is only needed in a multi-processor embedded system. It allows you to define multiple processors of the same type.

If for a derivative 'A' no processor is defined in the LSL file, the linker automatically creates a processor named 'A' of derivative 'A'. This is why for single-processor applications it is enough to specify the derivative in the LSL file.

The memory and bus definitions (optional)

Memory and bus definitions are used within the context of a derivative definition to specify internal memory and on-chip buses. In the context of a board specification the memory and bus definitions are used to define external (off-chip) memory and buses. Given the above definitions the linker can convert a logical address into an offset into an on-chip or off-chip memory device.

The board specification

The processor definition and memory and bus definitions together form a board specification. LSL provides language constructs to easily describe single-core and heterogeneous or homogeneous multi-core systems. The board specification describes all characteristics of your target board's system buses, memory devices, I/O sub-systems, and cores that are of interest to the linker. Based on the information provided in the board specification the linker can for each core:

- convert a logical address to an offset within a memory device

- locate sections in physical memory

- maintain an overall view of the used and free physical memory within the whole system while locating

The section layout definition (optional)

The optional section layout definition enables you to exactly control where input sections are located. Features are provided such as: the ability to place sections at a given address, to place sections in a given order, and to overlay code and/or data sections.

Example: Skeleton of a Linker Script File

A linker script file that defines a derivative "X" based on the TC1V1.3 architecture, its external memory and how sections are located in memory, may have the following skeleton:

```
architecture TC1V1.3
{
    // Specification of the TC1V1.3 core architecture.
    // Written by Altium.
}

derivative X  // derivative name is arbitrary
{
    // Specification of the derivative.
    // Written by Altium.
    core tc          // always specify the core
    {
        architecture = TC1V1.3;
    }

    bus fpi_bus      // internal bus
    {
        // maps to bus "fpi_bus" in "tc" core
    }
```

```
    // internal memory
}

processor spe        // processor name is arbitrary
{
    derivative = X;

    // You can omit this part, except if you use a
    // multi-core system.
}

memory ext_name
{
    // external memory definition
}

section_layout spe:tc:linear    // section layout
{
    // section placement statements

    // sections are located in address space 'linear'
    // of core 'tc' of processor 'spe'
}
```

Overview of LSL files delivered by Altium

Altium supplies the following LSL files in the directory `include.lsl`.

LSL file	Description
`tc_arch.lsl`	Defines the base architecture (TC) for all generic TriCore cores. It includes the files `base_address_groups.lsl`, `inttab.lsl` and `traptab.lsl`.
`tc_mc_arch.lsl`	Defines the base architecture (TC) for all multi-core TriCore cores.
`mcs_arch.lsl`	Defines the base architecture (MCS) for all MCS cores.
`base_address_groups.lsl`	Groups sections that belong to A0, A1, A8 or A9. It is included in the file `tc_arch.lsl`. See also linker option **--auto-base-register**
`inttab.lsl`	Defines the interrupt vector table. It is included in the file `tc_arch.lsl`.
`inttab{0|1|2}.lsl`	Defines a core specific interrupt vector table. It is included in derivative LSL files that have multi-core support.
`traptab.lsl`	Defines the trap vector table. It is included in the file `tc_arch.lsl`.
`traptab{0|1|2}.lsl`	Defines a core specific trap vector table. It is included in derivative LSL files that have multi-core support.

LSL file	Description

`tc1v1_3.lsl`
`tc1v1_3_1.lsl`
`tc1v1_6.lsl`
`tc1v1_6_x.lsl`
`tc1v1_6_2.lsl`
	Extends the base architecture for cores TC1V1.3, TC1V1.3.1, TC1V1.6,TC1V1.6.X or TC1V1.6.2. It includes the file `tc_arch.lsl` or `tc_mc_arch.lsl`.

derivative`.lsl`	Defines the derivative and defines a single processor (`spe`) or a multi-core processor (`mpe`). Contains a memory definition and section layout. It includes one of the files `tc`*version*`.lsl`. The selection of the derivative is based on your CPU selection (control program option **--cpu**). AURIX derivatives also include the file `bmhd.lsl`.
derivative_core`.lsl`	Defines a single *core* in a multi-core processor environment (`mpe`). Contains a memory definition and section layout. It includes one of the files `tc`*version*`.lsl`. The selection of the derivative is based on your CPU selection (control program option **--cpu**). The core is based on control program option **--lsl-core=***core*. AURIX derivatives also include the file `bmhd.lsl`.

`userdef13.lsl`
`userdef131.lsl`
`userdef16.lsl`
`userdef16x.lsl`
`userdef16x_`*core*`.lsl`
`userdef162.lsl`
`userdef162_`*core*`.lsl`
	Defines a user defined derivative for cores TC1V1.3, TC1V1.3.1, TC1V1.6, TC1V1.6.X or TC1V1.6.2 and defines a single processor for TC1V1.3, TC1V1.3.1 and TC1V1.6, and a multi-core processor for TC1V1.6.X and TC1V1.6.2, or a single core in a multi-core processor environment for TC1V1.6.X and TC1V1.6.2. `userdef16x.lsl`, `userdef16x_tc0.lsl`, `userdef162.lsl` and `userdef162_tc0.lsl` also include the file `bmhd.lsl`.

`template.lsl`	This file is used by Eclipse as a template for the project LSL file. It includes the file *derivative*`.lsl` or *derivative_core*`.lsl` based on your CPU selection. The CPU is specified by the __CPU__ macro.
`default.lsl`	Contains a default memory definition and section layout based on the tc1796b derivative. This file is used on a command line invocation of the tools, when no CPU is selected (no option **--cpu**). It includes the file `extmem.lsl`.
`extmem.lsl`	Template file with a specification of the external memory attached to the target processor.
`bmhd.lsl`	AURIX Boot Mode Header definition for BMHD0 .. BMHD3. See Section 7.8.9.2, *Boot Mode Headers*.

When you select to add a linker script file when you create a project in Eclipse, Eclipse makes a copy of the file `template.lsl` and names it "*project_name*`.lsl`". On the command line, the linker uses the file `default.lsl`, unless you specify another file with the linker option **--lsl-file** (**-d**).

7.8.5. The Architecture Definition

Although you will probably not need to write an architecture definition (unless you are building your own processor core) it helps to understand the Linker Script Language and how the definitions are interrelated.

Within an *architecture definition* the characteristics of a target processor core that are important for the linking process are defined. These include:

- space definitions: the logical address spaces and their properties

- bus definitions: the core local buses and I/O buses of the core architecture

- mappings: the address translations between logical address spaces, the connections between logical address spaces and buses and the address translations between buses

Address spaces

A logical address space is a memory range for which the core has a separate way to encode an address into instructions. Most microcontrollers and DSPs support multiple address spaces. For example, the TriCore's 32-bit linear address space encloses 16 24-bit sub-spaces and 16 14-bit sub-spaces. See also the section "*Memory Model*" in the *TriCore Architecture Manual*. Normally, the size of an address space is 2^N, with *N* the number of bits used to encode the addresses.

The relation of an address space with another address space can be one of the following:

- one space is a subset of the other. These are often used for "small" absolute or relative addressing.

- the addresses in the two address spaces represent different locations so they do not overlap. This means the core must have separate sets of address lines for the address spaces. For example, in Harvard architectures we can identify at least a code and a data memory space.

Address spaces (even nested) can have different minimal addressable units (MAU), alignment restrictions, and page sizes. All address spaces have a number that identifies the logical space (id). The following table lists the different address spaces for the architecture `TC` as defined in `tc_arch.lsl`.

Space	Id	MAU	Description	ELF sections
linear	1	8	Linear address space	.text*, .data*, .sdata*, .ldata*, .rodata*, .bss*, .sbss*, table, istack, ustack
abs24	2	8	Absolute 24-bit addressable space	
abs18	3	8	Absolute 18-bit addressable space	.zdata, .zrodata, .zbss
csa	4	8	Context Save Area	csa.*
pcp_code	8	16	PCP code	.pcptext
pcp_data	9	32	PCP data	.pcpdata

The following table lists the different address spaces for the architecture `TC` as defined in `tc_mc_arch.lsl` for multi-core processors.

Space	Id	MAU	Description	ELF sections
linear	1	8	Linear address space	.text*, .data*, .sdata*, .ldata*, .rodata*, .bss*, .sbss*, table, istack, ustack
abs24	2	8	Absolute 24-bit addressable space	
abs18	3	8	Absolute 18-bit addressable space	.zdata, .zrodata, .zbss
csa	4	8	Context Save Area	csa.*

The MCS, which is part of TriCore v1.6.x and v1.6.2 derivatives, such as TC27x, has one address space for architecture `MCS` as defined in `mcs_arch.lsl`

Space	Id	MAU	Description	ELF sections
mcs	1	8	MCS address space	.mcstext, .mcsdata

The MCS is described in a separate manual. See the *TASKING VX-toolset for MCS User Guide* for more information.

The TriCore architecture in LSL notation

The best way to write the architecture definition, is to start with a drawing. The following figure shows a part of the TriCore architecture:
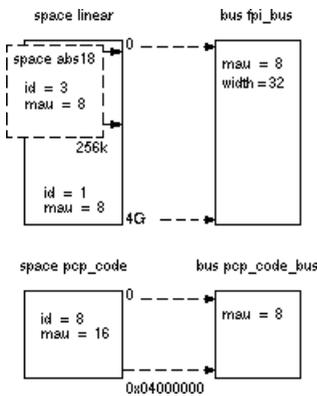
The figure shows three address spaces called `linear`, `abs18` and `pcp_code`. The address space `abs18` is a subset of the address space `linear`. All address spaces have attributes like a number that identifies the logical space (id), a MAU and an alignment. In LSL notation the definition of these address spaces looks as follows:

```
space linear
{
    id = 1;
    mau = 8;

    map (src_offset=0x00000000, dest_offset=0x00000000,
        size=4G, dest=bus:fpi_bus);
}

space abs18
{
    id = 3;
    mau = 8;

    map (src_offset=0x00000000, dest_offset=0x00000000,
        size=16k, dest=space:linear);
    map (src_offset=0x10000000, dest_offset=0x10000000,
```

```
            size=16k, dest=space:linear);
      map (src_offset=0x20000000, dest_offset=0x20000000,
            size=16k, dest=space:linear);
   //...
}

space pcp_code
{
      id = 8;
      mau = 16;
      map (src_offset=0x00000000, dest_offset=0,
            size=0x04000000, dest=bus:pcp_code_bus);
}
```

The keyword `map` corresponds with the arrows in the drawing. You can map:

- address space => address space

- address space => bus

- memory => bus (not shown in the drawing)

- bus => bus (not shown in the drawing)

Next the two internal buses, named `fpi_bus` and `pcp_code_bus` must be defined in LSL:

```
bus fpi_bus
{
      mau = 8;
      width = 32;  // there are 32 data lines on the bus
}

bus pcp_code_bus
{
      mau = 8;
      width = 8;
}
```

This completes the LSL code in the architecture definition. Note that all code above goes into the architecture definition, thus between:

```
architecture TC1V1.3
{
     // All code above goes here.
}
```

7.8.6. The Derivative Definition

Although you will probably not need to write a derivative definition (unless you are using multiple cores that both access the same memory device) it helps to understand the Linker Script Language and how the definitions are interrelated.

A *derivative* is the design of a processor, as implemented on a chip (or FPGA). It comprises one or more cores and on-chip memory. The derivative definition includes:

- core definition: an instance of a core architecture

- bus definition: the I/O buses of the core architecture

- memory definitions: internal (or on-chip) memory

Core

Each derivative must have at least one core and each core must have a specification of its core architecture. This core architecture must be defined somewhere in the LSL file(s).

```
core tc
{
    architecture = TC1V1.3;
}
```

A link task (resolving symbols from object files and libraries) is associated with one core in the LSL file(s). In a multi-core environment you can combine multiple cores with the same architecture into a single link task. This is done by importing one or more cores into a root core with an `import` statement. By importing a core the hardware resources of that core are made available to the link task associated with the core that contains the `import` statement. The imported cores share a single symbol namespace. The address spaces in each imported core must have a unique ID in the link task. For each imported core is specified that the space IDs of the imported core start at a specific offset. If writable sections for a core must be initialized by using the copy table of a different core, this is specified by a `copytable_space`. The following example is part of `tc27x.lsl` delivered with the product.

```
core tc0 // core 0
{
    architecture = TC1V1.6.X;
    space_id_offset = 100; // add 100 to all space IDs in
                           // the architecture definition
    copytable_space = vtc:linear; // use copytable from core vtc
}
core tc1 // core 1
{
    architecture = TC1V1.6.X;
    space_id_offset = 200; // add 200 to all space IDs in
                           // the architecture definition
    copytable_space = vtc:linear; // use copytable from core vtc
}

core tc2 // core 2
{
    architecture = TC1V1.6.X;
    space_id_offset = 300; // add 300 to all space IDs in
                           // the architecture definition
    copytable_space = vtc:linear; // use copytable from core vtc
}
```

```
core vtc
{
    architecture = TC1V1.6.X;
    import tc0; // add all address spaces of tc0 for linking
    import tc1; // add all address spaces of tc1 for linking
    import tc2; // add all address spaces of tc2 for linking
}
```

Bus

Each derivative can contain a bus definition for connecting external memory. In this example, the bus fpi_bus maps to the bus fpi_bus defined in the architecture definition of core tc:

```
bus fpi_bus
{
   mau = 8;
   width = 32;
   map (dest=bus:tc:fpi_bus, dest_offset=0, size=4G);
}
```

Memory

External memory is usually described in a separate memory definition, but you can specify on-chip memory for a derivative. For example:
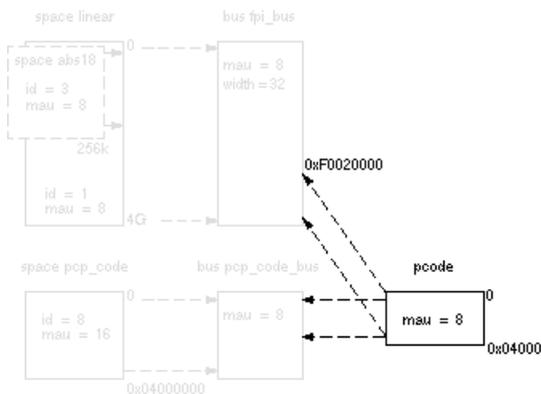
According to the drawing, the TriCore contains internal memory called pcode with a size 0x04000 (16 kB). This is physical memory which is mapped to the internal bus pcp_code_bus and to the fpi_bus, so both the tc unit and the PCP can access the memory:

```
memory pcode
{
    mau = 8;
    size = 16k;
    type = ram;
    map (dest=bus:tc:fpi_bus, dest_offset=0xF0020000,
```

```
          size=16k);
      map (dest=bus:tc:pcp_code_bus, size=16k);
}
```

This completes the LSL code in the derivative definition. Note that all code above goes into the derivative definition, thus between:

```
derivative X    // name of derivative
{
    // All code above goes here
}
```

7.8.7. The Processor Definition

The processor definition is only needed when you write an LSL file for a multi-processor embedded system. The processor definition explicitly instantiates a derivative, allowing multiple processors of the same type.

```
processor name
{
    derivative = derivative_name;
}
```

If no processor definition is available that instantiates a derivative, a processor is created with the same name as the derivative.

Altium defines a "single processor environment" (spe) in each `derivative.lsl` file. For example:

```
processor spe
{
    derivative = tc1796b;
}
```

For TriCore derivatives that have multiple processor cores, Altium defines a "multi-core processor environment" (mpe) in each `derivative.lsl` file. For example:

```
processor mpe
{
    derivative = tc27x;
}
```

7.8.8. The Memory Definition

Once the core architecture is defined in LSL, you may want to extend the processor with external (or off-chip) memory. You need to specify the location and size of the physical external memory devices in the target system.

The principle is the same as defining the core's architecture but now you need to fill the memory definition:

```
memory name
{
```

```
    // memory definitions
}
```
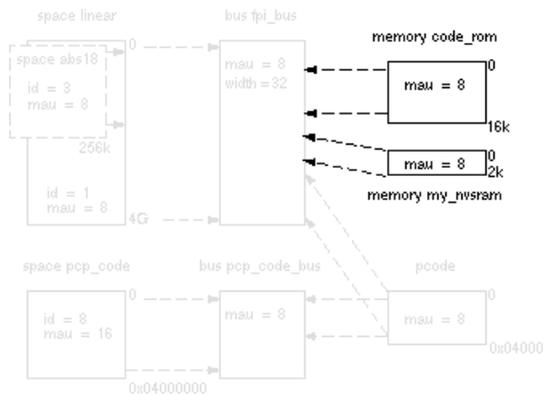

Suppose your embedded system has 16 kB of external ROM, named `code_rom` and 2 kB of external NVRAM, named `my_nvsram`. Both memories are connected to the bus `fpi_bus`. In LSL this looks like:

```
memory code_rom
{
    mau = 8;
    size = 16k;
    type = rom;
    map( dest=bus:spe:fpi_bus, dest_offset=0xa0000000, size=16k );
}

memory my_nvsram
{
    mau  = 8;
    size = 2k;
    type = nvram;
    map( dest=bus:spe:fpi_bus, dest_offset=0xc0000000, size=2k );
}
```

If you use a different memory layout than described in the LSL file supplied for the target core, you can specify this in Eclipse or you can specify this in a separate LSL file and pass both the LSL file that describes the core architecture and your LSL file that contains the memory specification to the linker.

To add memory using Eclipse

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Memory**.

    In the right pane the Memory page appears.

3. Open the **Memory** tab and click on the **Add...** button.

The Add new memory dialog appears.

4.   Enter the memory name (for example `my_nvsram`), type (for example `nvram`) and size.

5.   Click on the **Add...** button.

The Add new mapping dialog appears.

6.   You have to specify at least one mapping. Enter the mapping name (optional), address, size and destination and click **OK**.

The new mapping is added to the list of mappings.

7.   Click **OK**.

The new memory is added to the list of memories (user memory).

8.   Click **OK** to close the Properties dialog.

The updated settings are stored in the project LSL file.

If you make changes to the on-chip memory as defined in the architecture LSL file, the memory is copied to your project LSL file and the line `#define __REDEFINE_ON_CHIP_ITEMS` is added. If you remove all the on-chip memory from your project LSL file, also make sure you remove this define.

7.8.9. The Section Layout Definition: Locating Sections

Once you have defined the internal core architecture and optional memory, you can actually define where your application must be located in the physical memory.

During compilation, the compiler divides the application into sections. Sections have a name, an indication (section type) in which address space it should be located and attributes like writable or read-only.

In the section layout definition you can exactly define how input sections are placed in address spaces, relative to each other, and what their absolute run-time and load-time addresses will be.

Example: section propagation through the toolset

To illustrate section placement, the following example of a C program (`bat.c`) is used. The program saves the number of times it has been executed in battery back-upped memory, and prints the number.

```
#define BATTERY_BACKUP_TAG  0xa5f0
#include <stdio.h>

int  uninitialized_data;
int  initialized_data = 1;
#pragma section all "non_volatile"
#pragma noclear
int  battery_backup_tag;
int  battery_backup_invok;
#pragma clear
#pragma section all
```

```
void main (void)
{
    if (battery_backup_tag != BATTERY_BACKUP_TAG )
    {
        // battery back-upped memory area contains invalid data
        // initialize the memory
        battery_backup_tag = BATTERY_BACKUP_TAG;
        battery_backup_invok = 0;
    }
    printf( "This application has been invoked %d times\n",
            battery_backup_invok++);
}
```

The compiler assigns names and attributes to sections. With the `#pragma section` the compiler's default section naming convention is overruled and a section with the name `non_volatile` is defined. In this section the battery back-upped data is stored.

By default the compiler creates a section with the name ".zbss.bat" of section type `data` to store uninitialized data objects. The section prefix ".zbss" tells the linker to locate the section in address space `abs18` and that the section content should be filled with zeros at startup.

As a result of the `#pragma section all "non_volatile"`, the data objects between the pragma pair are placed in a section with the name ".zbss.non_volatile". Note that ".zbss" sections are cleared at startup. However, battery back-upped sections should not be cleared and therefore we used `#pragma noclear`.

Section placement

The number of invocations of the example program should be saved in non-volatile (battery back-upped) memory. This is the memory `my_nvsram` from the example in Section 7.8.8, *The Memory Definition*.

To control the locating of sections, you need to write one or more section definitions in the LSL file. At least one for each address space where you want to change the default behavior of the linker. In our example, we need to locate sections in the address space `abs18`:

```
section_layout ::abs18
{
    select "ELF sections";
    // Section placement statements
}
```

The space, in this case `abs18`, and the *ELF sections* must be a valid combination from the table in Section 7.8.5, *The Architecture Definition*.

To locate sections, you must create a group in which you select sections from your program. For the battery back-up example, we need to define one group, which contains the section .zbss.non_volatile. All other sections are located using the defaults specified in the architecture definition. Section .zbss.non_volatile should be placed in non-volatile ram. To achieve this, the run address refers to our non-volatile memory called `my_nvsram`.

```
group ( ordered, run_addr = mem:my_nvsram )
{
      select ".zbss.non_volatile";
}
```

This completes the LSL file for the sample architecture and sample program. You can now invoke the linker with this file and the sample program to obtain an application that works for this architecture.

For a complete description of the Linker Script Language, refer to Chapter 16, *Linker Script Language (LSL)*.

7.8.9.1. Locating Private Sections in ROM

For TriCore derivatives that have multi-core support, such as the TC27x, private sections are by default located in core-local RAM. If however all core-local RAM is used, you can tell the linker to locate private sections in ROM. You can do this by adding the keywords `nocopy, attributes=rx` to the group specification in LSL. See also Section 1.4, *Multi-Core Support*.

The following example shows the function `main()` in `main.c` that calls function `p0()` that is marked as `__private0` in `private0.c`.

```
/* main.c */
extern void __private0 p0( void );
extern int i;
int main( void )
{
    p0();
    return i;
}

/* private0.c */
int i;

void __private0 p0( void )
{
 i++;
}
```

To specify that the section `.text.private0.private0.p0` must be located in ROM instead of core-local RAM, you can specify the following LSL part:

```
// nocopy.lsl

section_layout mpe:tc0:code
{
    group PSPR ( run_addr = mem:mpe:pflash0, nocopy, attributes=rx )
    {
        select ".text.private0.private0.p0";
    }
}
```

The keyword `nocopy` specifies that the section is not copied from ROM to RAM at program startup and `attributes=rx` marks the section read-only and executable.

After the following invocation on the command line, you can inspect the resulting map file to see the results.

```
cctc -Ctc27x main.c private0.c -Wl-dnocopy.lsl
```

Part of map file:

```
+ Space mpe:tc0:code (MAU = 8bit)


+-----------------------------------------------------------------------
| Chip        | Group | Section                          | Size (MAU) | ...
|=======================================================================
| mpe:pflash0 | PSPR  | .text.private0.private0.p0 (2) | 0x0000000c | ...
+-----------------------------------------------------------------------
```

7.8.9.2. Boot Mode Headers

The linker and LSL has support for the AURIX Boot Mode Headers BMHD0 .. BMHD3. For detailed information, for example for the TC27x, about Boot Mode Headers, Startup Software (SSW) and Bootstrap loaders refer to *chapter 5 TC27x BootROM Content* in the *AURIX TC27x User's Manual*.

Altium delivers AURIX LSL files, such as `tc27x.lsl`, in the `include.lsl` directory of the product. Support for the boot mode headers is present in a separate LSL file, `bmhd.lsl`, which is included in the TriCore derivative LSL files.

> Note that these system LSL files are not to be changed.

Boot Mode Header configurations

A Boot Mode Header (BMHD) is defined with a `struct` statement in `bmhd.lsl`. Part of the BMHD structure is the Boot Mode Index (BMI), which holds information, for example, about the start-up mode of the device.

You can configure each of the Boot Mode Headers BMHD0 .. BMHD3. The system LSL file `bmhd.lsl` contains macros for the configuration. This system LSL file is not to be changed. However, you can use macros in your project LSL file or you can use Eclipse to do this for you. When you make changes in Eclipse, macros are added to the project's LSL file. The following configurations are possible for each Boot Mode Header:

Boot Mode Header configuration	Description
Reserved Boot Mode Header	Only the memory areas for the Boot Mode Headers are reserved. This means it will not be flashed. This does not result in any settings in the project's LSL files, because this is the default in `bmhd.lsl`.
Generate Boot Mode Header	The Boot Mode Header is generated with the values defined in Eclipse.

Boot Mode Header configuration	Description
Invalidate Boot Mode Header	The Boot Mode Header is initialized with zeros (invalid data). This ensures that the processor will not use this Boot Mode Header.
Disable Boot Mode Header	The Boot Mode Header is not generated and no memory area is reserved. This setting is not supported in Eclipse. This is for users who want to define everything themselves. To do this, set the macro `__BMHDn_CONFIG` to `__BMHD_DISABLE` in the project's LSL file.

Note that you cannot set all Boot Mode Headers to "invalidate". The linker generates an error because this combination might brick the device.

To configure Boot Mode headers using macros in LSL

If you do not use Eclipse, you can configure the Boot Mode Headers in your project LSL file. The following is an example of a configuration in project LSL file `myproject.lsl`.

```
//** start of myproject.lsl
//** define macros for Boot Mode Headers
#define __BMHD0_CONFIG      __BMHD_GENERATE
#define __BMHD0_STADABM     _START
#define __BMHD0_BMI         __BMHD_BMI_PINDIS|__BMHD_BMI_HWCFG_ABM
#define __BMHD0_CHKSTART    addressof(mem:flash)
#define __BMHD0_CHKEND      addressof(mem:flash) + sizeof(mem:flash)

//** include the TC27x system LSL file
#include "tc27x.lsl"
```

This example tells to generate Boot Mode Header BMHD0, use the Alternate Boot Mode (ABM) as start-up mode for the device and calculate a checksum over the specified flash memory range.

You can use the following macros to control the Boot Mode Headers. The *n* in the macro name can be 0, 1, 2 or 3.

Macro	Default value	Description
`__BMHDnCONFIG`	`__BMHD_RESERVE`	The Boot Mode Header Configuration. This macro must be defined with one of the following values: `__BMHD_RESERVE` Only reserve memory areas for the Boot Mode Header. `__BMHD_GENERATE` Generate the Boot Mode Header with the values defined with the macros in this table. `__BMHD_INVALIDATE` Initialize the Boot Mode Header with zeros (invalid data). `__BMHD_DISABLE` Disable the Boot Mode Header generation and reservation.
`__BMHDnSTADABM`	`_START`	Defines the user code start address.
`__BMHDnBMI`	`__BMHD_BMI_HWCFG_INTERNAL`	Defines the BMI value, the default is the factory settings. The various fields are described with macros in the table below.
`__BMHDnCHKSTART`	`"_START"`	Defines the start address of the memory range to calculate the checksum of.
`__BMHDnCHKEND`	`"_START" + 4`	Defines the end address of the memory range to calculate the checksum of. The specified address is the first address after the last byte that is part of the checksum.

Note that for the checksum memory range start address and end address you can specify labels, such as `"_START"`, absolute addresses, LSL built-in functions or linker labels.

The following macros are available for composing the value of the BMI field. These macros are the same for each Boot Mode Header.

BMI field macro	Value	Description
`__BMHD_BMI_PINDIS`	0x0008	Disable mode selection by HWCFG pins.
`__BMHD_BMI_HWCFG_INTERNAL`	0x0070	Hardware start-up mode: internal start from flash
`__BMHD_BMI_HWCFG_ABM`	0x0060	Hardware start-up mode: Alternate Boot Mode (ABM)
`__BMHD_BMI_HWCFG_GENERIC_BSL`	0x0040	Hardware start-up mode: Generic Bootstrap Loader
`__BMHD_BMI_HWCFG_ASC_BSL`	0x0040	Hardware start-up mode: ASC Bootstrap Loader
`__BMHD_BMI_LCL0LSEN`	0x0040	Lockstep for CPU 0
`__BMHD_BMI_LCL1LSEN`	0x0030	Lockstep for CPU 1

Additionally, the following macro is defined in `bmhd.lsl` to control the Boot Mode Header alignment:

```
__BMHD_ALIGN( address, alignment )
```

Both the *address* and *alignment* arguments are expressions. Note that the expressions simply evaluate to a number. So, the address and alignment are also (integer) numbers. In practice those numbers always represent MAUs, though. When the *alignment* argument is a positive value the result will be the first aligned address higher than or equal to the specified *address* argument. When the *alignment* argument is a negative value the result will be the first aligned address lower than or equal to the specified *address* argument.

To configure Boot Mode Headers using Eclipse

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Memory**.

    In the right pane the Memory page appears.

3. Open the **Boot Mode Headers** tab.

    Four group boxes are present, one for each Boot Mode Header.
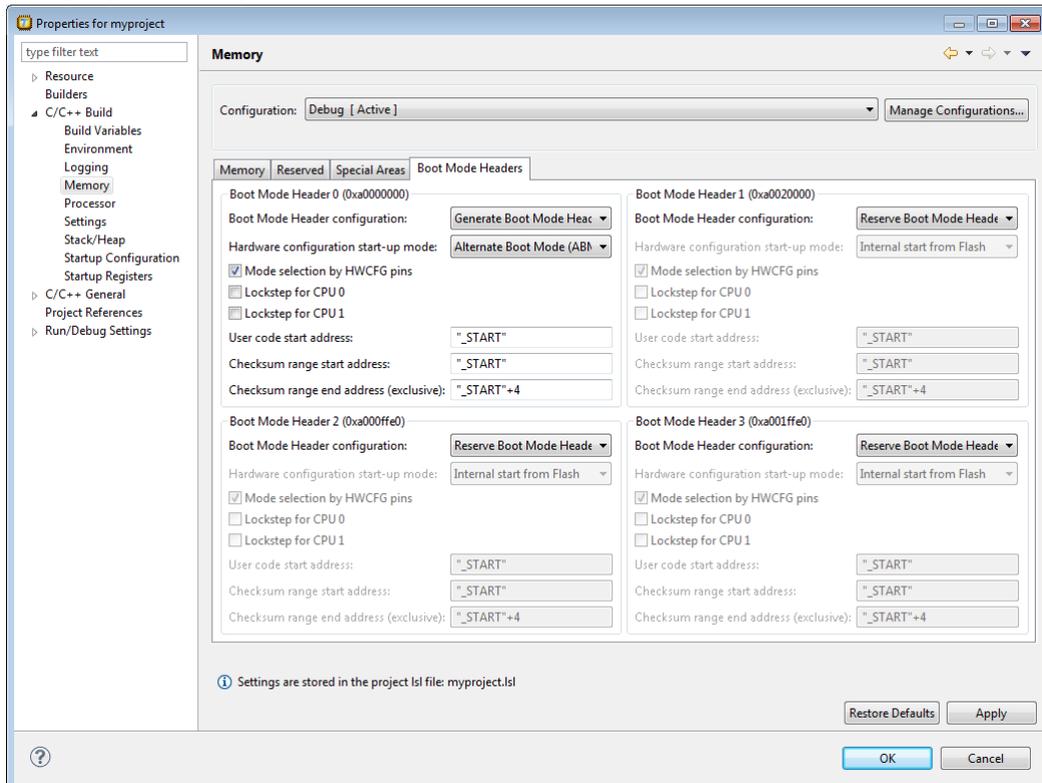
4. Select a **Boot Mode Header configuration**.

   *When you select **Generate Boot Mode Header**, you can optionally specify additional information as specified below.*

5. (Optionally) Specify one or more of the following options:

   • **Hardware configuration start-up mode**

   Here you can select the start-up mode configuration by bits `HWCFG[5:4]` of the BMI. Select one of:

   • Internal start from Flash

   • Alternate Boot Mode (ABM)

   • Generic Bootstrap Loader

   • ASC Bootstrap Loader

   • **Mode by HWCFG pins**

   When you disable this option, the Mode selection by HWCFG pins is disabled. This corresponds to setting the `PINDIS` field of the BMI to 1. This option is enabled by default (`PINDIS=0`).

   • **Lockstep for CPU 0 / 1**

   This is the Lockstep Comparator Logic control for CPU 0 and/or CPU 1. These options correspond to the `LCL0LSEN` and `LCL1LSEN` fields of the BMI. By default these options are disabled.

6. For the **Alternate Boot Mode (ABM)** start-up mode, specify the following information:

   • **User code start address**

   Enter the start address of your application. This field corresponds to the start address alternate boot mode field (`STADABM`) of the Boot Mode Header (BMHD) structure.

   • **Checksum range start address / end address (exclusive)**

   Specify the memory address range that needs to be checked. A crc32 checksum is calculated over the specified address range. Note that in the end address you specify here is not part of the range. These fields correspond to the `ChkStart` and `ChkEnd + 4` fields of the Boot Mode Header (BMHD) structure.

   Example

   You can specify labels, such as `"_START"`, absolute addresses, LSL built-in functions or linker labels. For example to use the start and end address of a group named `crc32_bmhd0`, you can specify:

   Checksum range start address: `"_lc_gb_crc32_bmhd0"`

   Checksum range end address (exclusive): `"_lc_ge_crc32_bmhd0"`

259

or:

Checksum range start address: `addressof(group:crc32_bmhd0)`

Checksum range end address (exclusive): `addressof(group:crc32_bmhd0)+`
`sizeof(group:crc32_bmhd0)`

7. Click **OK** to close the Properties dialog.

    The updated settings are stored in the project LSL file.

7.9. Linker Labels

The linker creates labels that you can use to refer to from within the application software. Some of these labels are real labels at the beginning or the end of a section. Other labels have a second function, these labels are used to address generated data in the locating phase. The data is only generated if the label is used.

Linker labels are labels starting with `_lc_`. The linker assigns addresses to the following labels when they are referenced:

Label	Description
`_lc_ub_`*name* `_lc_b_`*name*	Begin of section *name*. Also used to mark the lowest address of the stack or heap or copy table.
`_lc_ue_`*name* `_lc_e_`*name*	End of section *name*. Also used to mark the highest address of the stack or heap.
`_lc_cb_`*name*	Start address of an overlay section in ROM.
`_lc_ce_`*name*	End address of an overlay section in ROM.
`_lc_gb_`*name*	Begin of group *name*. This label appears in the output file even if no reference to the label exists in the input file.
`_lc_ge_`*name*	End of group *name*. This label appears in the output file even if no reference to the label exists in the input file.
`_lc_s_`*name*	Variable *name* is mapped through memory in shared memory situations.
`_lc_t_`*core_name*	Variable *name* in the specified *core* is mapped to the address space of the referred section. This way you can refer to a variable on a specific core on the same processor.

The linker only allocates space for the stack and/or heap when a reference to either of the section labels exists in one of the input object files.

Additionally, the linker script file defines the following symbols:

Symbol	Description
`_lc_cp`	Start of copy table. Same as `_lc_ub_table`. The copy table gives the source and destination addresses of sections to be copied. This table will be generated by the linker only if this label is used.
`_lc_bh`	Begin of heap. Same as `_lc_ub_heap`.
`_lc_eh`	End of heap. Same as `_lc_ue_heap`.

If you want to use linker labels in your C source for sections that have a dot (.) in the name, you have to replace all dots by underscores.

Example: refer to a label with section name with dots from C

Suppose the C source file `foo.c` contains the following:

```
int myfunc(int a)
{
    /* some source lines */
    return 1;
}
```

This results in a section with the name `.text.foo.myfunc`.

In the following source file `main.c` all dots of the section name are replaced by underscores:

```
#include <stdio.h>
extern char _lc_ub__text_foo_myfunc[];

void main(void)
{
    printf("The function myfunc is located at %x\n",
            _lc_ub__text_foo_myfunc);
}
```

Example: refer to a PCP variable from TriCore C source

When memory is shared between two or more cores, for instance TriCore and PCP, the addresses of variables (or functions) on that memory may be different for the cores. For the TriCore the variable will be defined and you can access it in the usual way. For the PCP, when you would use the variable directly in your TriCore source, this would use an incorrect address (PCP address). The linker can map the address of the variable from one space to another, if you prefix the variable name with **`_lc_s_`**.

When a symbol `foo` is defined in a PCP assembly source file, by default it gets the symbol name `foo`. To use this symbol from a TriCore C source file, write:

```
extern long _lc_s_foo;

void main(int argc, char **argv)
{
```

```
    _lc_s_foo = 7;
}
```

Example: refer to an MCS variable from TriCore C source

From within the TriCore source you can access MCS variables. The same symbol name can be defined in different MCS cores. To uniquely select a variable from a core, you prefix the variable name with **_lc_t_***core*_. When the linker sees the **_lc_t_**, it removes the linker label prefix, and the core name prefix. The remainder is a symbol name, that has to be found inside the *core*.

For example, when a symbol count is defined in assembly sources of two different MCS cores, you can access them from a TriCore C source file as follows:

```
extern int _lc_t_mcs00_count; /* variable count in mcs00 */
extern int _lc_t_mcs01_count; /* variable count in mcs01 *

void main(int argc, char **argv)
{
    _lc_mcs00_count = 0;
    _lc_mcs01_count = 1;
}
```

Example: refer to the stack

Suppose in an LSL file a stack section is defined with the name "ustack" (with the keyword stack). You can refer to the begin and end of the stack from your C source as follows:

```
#include <stdio.h>
extern char _lc_ub_ustack[];
extern char _lc_ue_ustack[];
void main()
{
  printf( "Size of stack is %d\n",
          _lc_ue_ustack - _lc_ub_ustack );
}
```

From assembly you can refer to the end of the stack with:

```
   .extern _lc_ue_ustack   ; end of user stack
```

See Section 1.11.1, *Calling Convention* and section *2.2.2 Stack Frame Management* in the TriCore EABI for more information about the stack.

7.10. Generating a Map File

The map file is an additional output file that contains information about the location of sections and symbols. You can customize the type of information that should be included in the map file.

To generate a map file

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Map File**.

4. Enable the option **Generate XML map file format (.mapxml) for map file viewer**.

5. (Optional) Enable the option **Generate map file (.map)**.

6. (Optional) Enable the options to include that information in the map file.

Example on the command line (Windows Command Prompt)

The following command generates the map file `test.map`:

```
ltc --map-file test.o
```

With this command the map file `test.map` is created.

See Section 14.2, *Linker Map File Format*, for an explanation of the format of the map file.

7.11. Linker Error Messages

The linker reports the following types of error messages in the Problems view of Eclipse.

F (Fatal errors)

After a fatal error the linker immediately aborts the link/locate process.

E (Errors)

Errors are reported, but the linker continues linking and locating. No output files are produced unless you have set the linker option **--keep-output-files**.

W (Warnings)

Warning messages do not result into an erroneous output file. They are meant to draw your attention to assumptions of the linker for a situation which may not be correct. You can control warnings in the **C/C++ Build » Settings » Tool Settings » Linker » Diagnostics** page of the **Project » Properties for** menu (linker option **--no-warnings**).

I (Information)

Verbose information messages do not indicate an error but tell something about a process or the state of the linker. To see verbose information, use the linker option **--verbose**.

S (System errors)

System errors occur when internal consistency checks fail and should never occur. When you still receive the system error message

```
S6##: message
```

please report the error number and as many details as possible about the context in which the error occurred.

Display detailed information on diagnostics

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

   The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

   A popup menu appears.

3. Select **Detailed Diagnostics Info**.

   A dialog box appears with additional information.

On the command line you can use the linker option **--diag** to see an explanation of a diagnostic message:

```
ltc --diag=[format:]{all | number,...]
```

Chapter 8. Using the Utilities

The TASKING VX-toolset for TriCore comes with a number of utilities:

cctc   A control program. The control program invokes all tools in the toolset and lets you quickly generate an absolute object file from C and/or assembly source input files. Eclipse uses the control program to call the compiler, assembler and linker.

mktc   A utility program to maintain, update, and reconstruct groups of programs. The make utility looks whether files are out of date, rebuilds them and determines which other files as a consequence also need to be rebuilt.

amk   The make utility which is used in Eclipse. It supports parallelism which utilizes the multiple cores found on modern host hardware.

eclipsec   The Eclipse console utility. You can use it to perform a headless build or generate makefiles from the command line without starting the IDE.

artc   An archiver. With this utility you create and maintain library files with relocatable object modules (`.o`) generated by the assembler.

hldumptc   A high level language (HLL) object dumper. With this utility you can dump information about an absolute object file (`.elf`). Key features are a disassembler with HLL source intermixing and HLL symbol display and a HLL symbol listing of static and global symbols.

elfpatch   A utility to change one or more section names or modify data references within a relocatable ELF object file (`.o`) or library (`.a`).

expiretc   A utility to limit the size of the cache by removing all files older than a few days or by removing older files until the total size of the cache is smaller than a specified size.

proftool   A utility used by the TASKING Profiler perspective in Eclipse to display profiling information.

ichk   A utility to check the integrity of an application. This utility checks access restrictions on memory in light of ASIL specifications and MPU configuration.

8.1. Control Program

The control program is a tool that invokes all tools in the toolset for you. It provides a quick and easy way to generate the final absolute object file out of your C/C++ sources without the need to invoke the compiler, assembler and linker manually.

Eclipse uses the control program to call the C++ compiler, C compiler, assembler and linker, but you can call the control program from the command line. The invocation syntax is:

cctc [[*option*]... [*file*]...]...

Recognized input files

- Files with a `.cc`, `.cxx` or `.cpp` suffix are interpreted as C++ source programs and are passed to the C++ compiler.

- Files with a `.c` suffix are interpreted as C source programs and are passed to the compiler.

- Files with a `.asm` suffix are interpreted as hand-written assembly source files which have to be passed to the assembler.

- Files with a `.src` suffix are interpreted as compiled assembly source files. They are directly passed to the assembler.

- Files with a `.a` suffix are interpreted as library files and are passed to the linker.

- Files with a `.o` suffix are interpreted as object files and are passed to the linker.

- Files with a `.out` suffix are interpreted as linked object files and are passed to the locating phase of the linker. The linker accepts only one `.out` file in the invocation.

- Files with a `.lsl` suffix are interpreted as linker script files and are passed to the linker.

Options

The control program accepts several command line options. If you specify an unknown option to the control program, the control program looks if it is an option for a specific tool. If so, it passes the option directly to the tool. However, it is recommended to use the control program options **--pass-*** (**-Wcp**, **-Wc**, **-Wa**, **-Wl**) to pass arguments directly to tools.

For a complete list and description of all control program options, see Section 10.6, *Control Program Options*.

Example with verbose output

```
cctc --verbose test.c
```

The control program calls all tools in the toolset and generates the absolute object file `test.elf`. With option **--verbose** (**-v**) you can see how the control program calls the tools:

```
+ "path\ctc" -o cc3248a.src test.c
+ "path\astc" -o cc3248b.o cc3248a.src
+ "path\ltc" -o test.elf -dextmem.lsl -ddefault.lsl --map-file
    cc3248b.o "-Lpath\lib\tc1" -lc -lfp -lrt
```

The control program produces unique filenames for intermediate steps in the compilation process (such as `cc3248a.src` and `cc3248b.o` in the example above) which are removed afterwards, unless you specify command line option **--keep-temporary-files** (**-t**).

Example with argument passing to a tool

```
cctc --pass-compiler=-Oc test.c
```

The option **-Oc** is directly passed to the compiler.

8.2. Make Utility mktc

If you are working with large quantities of files, or if you need to build several targets, it is rather time-consuming to call the individual tools to compile, assemble, link and locate all your files.

You save already a lot of typing if you use the control program and define an options file. You can even create a batch file or script that invokes the control program for each target you want to create. But with these methods all files are completely compiled, assembled and linked to obtain the target file, even if you changed just one C source. This may demand a lot of (CPU) time on your host.

The make utility **mktc** is a tool to maintain, update, and reconstruct groups of programs. The make utility looks which files are out-of-date and only recreates these files to obtain the updated target.

Make process

In order to build a target, the make utility needs the following input:

• the target it should build, specified as argument on the command line

• the rules to build the target, stored in a file usually called `makefile`

In addition, the make utility also reads the file `mktc.mk` which contains predefined rules and macros. See Section 8.2.2, *Writing a Makefile*.

The `makefile` contains the relationships among your files (called *dependencies*) and the commands that are necessary to create each of the files (called *rules*). Typically, the absolute object file (`.elf`) is updated when one of its dependencies has changed. The absolute file depends on `.o` files and libraries that must be linked together. The `.o` files on their turn depend on `.src` files that must be assembled and finally, `.src` files depend on the C source files (`.c`) that must be compiled. In the `makefile` this looks like:

```
test.src : test.c                    # dependency
         ctc test.c                   # rule

test.o   : test.src
         astc test.src

test.elf : test.o
         ltc test.o -o test.elf --map-file -lc -lfp -lrt
```

You can use any command that is valid on the command line as a rule in the `makefile`. So, rules are not restricted to invocation of the toolset.

Example

To build the target `test.elf`, call **mktc** with one of the following lines:

```
mktc test.elf
```

```
mktc -fmymake.mak test.elf
```

By default the make utility reads the file `makefile` so you do not need to specify it on the command line. If you want to use another name for the makefile, use the option **-f**.

If you do not specify a target, **mktc** uses the first target defined in the makefile. In this example it would build `test.src` instead of `test.elf`.

Based on the sample invocation, the make utility now tries to build `test.elf` based on the makefile and performs the following steps:

1. From the makefile the make utility reads that `test.elf` depends on `test.o`.

2. If `test.o` does not exist or is out-of-date, the make utility first tries to build this file and reads from the makefile that `test.o` depends on `test.src`.

3. If `test.src` does exist, the make utility now creates `test.o` by executing the rule for it: `astc test.src`.

4. There are no other files necessary to create `test.elf` so the make utility now can use `test.o` to create `test.elf` by executing the rule: `ltc test.o -o test.elf ...`

The make utility has now built `test.elf` but it only used the assembler to update `test.o` and the linker to create `test.elf`.

If you compare this to the control program:

```
cctc test.c
```

This invocation has the same effect but now *all files* are recompiled (assembled, linked and located).

8.2.1. Calling the Make Utility

You can only call the make utility from the command line. The invocation syntax is:

mktc [[*option*]... [*target*]... [*macro=def*]...]

For example:

```
mktc test.elf
```

target	You can specify any target that is defined in the makefile. A target can also be one of the intermediate files specified in the makefile.

macro=def	Macro definition. This definition remains fixed for the **mktc** invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is inherited by subordinate **mktc**'s but act as an environment variable for these. That is, depending on the **-e** setting, it may be overridden by a makefile definition.
option	For a complete list and description of all make utility options, see Section 10.7, *Make Utility Options*.

Exit status

The make utility returns an exit status of 1 when it halts as a result of an error. Otherwise it returns an exit status of 0.

8.2.2. Writing a Makefile

In addition to the standard makefile makefile, the make utility always reads the makefile mktc.mk before other inputs. This system makefile contains implicit rules and predefined macros that you can use in the makefile makefile.

With the option **-r** (Do not read the mktc.mk file) you can prevent the make utility from reading mktc.mk.

The default name of the makefile is makefile in the current directory. If you want to use another makefile, use the option **-f**.

The makefile can contain a mixture of:

* targets and dependencies

* rules

* macro definitions or functions

* conditional processing

* comment lines

* include lines

* export lines

To continue a line on the next line, terminate it with a backslash (****):

```
# this comment line is continued\
on the next line
```

If a line must end with a backslash, add an empty macro:

```
# this comment line ends with a backslash \$(EMPTY)
# this is a new line
```

8.2.2.1. Targets and Dependencies

The basis of the makefile is a set of targets, dependencies and rules. A target entry in the makefile has the following format:

```
target ... : [dependency ...] [; rule]
        [rule]
          ...
```

Target lines must always start at the beginning of a line, leading white spaces (tabs or spaces) are not allowed. A target line consists of one or more targets, a semicolon and a set of files which are required to build the target (*dependencies*). The target itself can be one or more filenames or symbolic names:

```
all:                    demo.elf final.elf

demo.elf final.elf:     test.o demo.o final.o
```

You can now can specify the target you want to build to the make utility. The following three invocations all have the same effect:

```
mktc
mktc all
mktc demo.elf final.elf
```

If you do *not* specify a target, the first target in the makefile (in this example `all`) is built. The target `all` depends on `demo.elf` and `final.elf` so the second and third invocation have the same effect and the files `demo.elf` and `final.elf` are built.

You can normally use colons to denote drive letters. The following works as intended:

```
c:foo.o : a:foo.c
```

If a target is defined in more than one target line, the dependencies are added to form the target's complete dependency list:

```
all:  demo.elf   # These two lines are equivalent with:
all:  final.elf  # all: demo.elf final.elf
```

Special targets

There are a number of special targets. Their names begin with a period.

Target	Description
.DEFAULT	If you call the make utility with a target that has no definition in the makefile, this target is built.
.DONE	When the make utility has finished building the specified targets, it continues with the rules following this target.
.IGNORE	Non-zero error codes returned from commands are ignored. Encountering this in a makefile is the same as specifying the option **-i** on the command line.
.INIT	The rules following this target are executed before any other targets are built.

Target	Description
`.PRECIOUS`	Dependency files mentioned for this target are never removed. Normally, if a command in a rule returns an error or when the target construction is interrupted, the make utility removes that target file. You can use the option **-p** on the command line to make all targets precious.
`.SILENT`	Commands are not echoed before executing them. Encountering this in a makefile is the same as specifying the option **-s** on the command line.
`.SUFFIXES`	This target specifies a list of file extensions. Instead of building a completely specified target, you now can build a target that has a certain file extension. Implicit rules to build files with a number of extensions are included in the system makefile `mktc.mk`. If you specify this target with dependencies, these are added to the existing `.SUFFIXES` target in `mktc.mk`. If you specify this target without dependencies, the existing list is cleared.

8.2.2.2. Makefile Rules

A line with leading white space (tabs or spaces) is considered as a rule and associated with the most recently preceding dependency line. A *rule* is a line with commands that are executed to build the associated target. A target-dependency line can be followed by one or more rules.

```
final.src : final.c              # target and dependency
            move test.c final.c  # rule1
            ctc final.c          # rule2
```

You can precede a rule with one or more of the following characters:

@   does not echo the command line, except if **-n** is used.

-   the make utility ignores the exit code of the command. Normally the make utility stops if a non-zero exit code is returned. This is the same as specifying the option **-i** on the command line or specifying the special `.IGNORE` target.

+   The make utility uses a shell or Windows command prompt (`cmd.exe`) to execute the command. If the '+' is not followed by a shell line, but the command is an MS-DOS command or if redirection is used (<, |, >), the shell line is passed to `cmd.exe` anyway.

You can force **mktc** to execute multiple command lines in one shell environment. This is accomplished with the token combination '**;**'. For example:

```
cd c:\Tasking\bin ;\
mktc -V
```

Note that the ';' must always directly be followed by the '\' token. Whitespace is not removed when it is at the end of the previous command line or when it is in front of the next command line. The use of the ';' as an operator for a command (like a semicolon ';' separated list with each item on one line) and the '\' as a layout tool is not supported, unless they are separated with whitespace.

Inline temporary files

The make utility can generate inline temporary files. If a line contains **<<***LABEL* (no whitespaces!) then all subsequent lines are placed in a temporary file until the line *LABEL* is encountered. Next, **<<***LABEL* is replaced by the name of the temporary file. For example:

```
ltc -o $@ -f <<EOF
     $(separate "\n" $(match .o $!))
     $(separate "\n" $(match .a $!))
     $(LKFLAGS)
EOF
```

The three lines between `<<EOF` and `EOF` are written to a temporary file (for example `mkce4c0a.tmp`), and the rule is rewritten as: `ltc -o $@ -f mkce4c0a.tmp`.

Suffix targets

Instead of specifying a specific target, you can also define a general target. A general target specifies the rules to generate a file with extension `.ex1` to a file with extension `.ex2`. For example:

```
.SUFFIXES:  .c
.c.o    :
            cctc -c $<
```

Read this as: to build a file with extension `.o` out of a file with extension `.c`, call the control program with **-c $<**. **$<** is a predefined macro that is replaced with the name of the current dependency file. The special target `.SUFFIXES:` is followed by a list of file extensions of the files that are required to build the target.

Implicit rules

Implicit rules are stored in the system makefile `mktc.mk` and are intimately tied to the `.SUFFIXES` special target. Each dependency that follows the `.SUFFIXES` target, defines an extension to a filename which must be used to build another file. The implicit rules then define how to actually build one file from another. These files share a common basename, but have different extensions.

If the specified target on the command line is not defined in the makefile or has not rules in the makefile, the make utility looks if there is an implicit rule to build the target.

Example:

```
LIB  =     -lc -lfp -lrt        # macro

prog.elf:  prog.o sub.o
     ltc   prog.o sub.o $(LIB) -o prog.elf

prog.o:    prog.c inc.h
     ctc   prog.c
     astc  prog.src

sub.o:     sub.c inc.h
     ctc   sub.c
     astc  sub.src
```

This makefile says that `prog.elf` depends on two files `prog.o` and `sub.o`, and that they in turn depend on their corresponding source files (`prog.c` and `sub.c`) along with the common file `inc.h`.

The following makefile uses implicit rules (from `mktc.mk`) to perform the same job.

```
LDFLAGS = -lc -lfp -lrt        # macro used by implicit rules
prog.elf: prog.o sub.o         # implicit rule used
prog.o: prog.c inc.h           # implicit rule used
sub.o:  sub.c inc.h            # implicit rule used
```

8.2.2.3. Macro Definitions

A *macro* is a symbol name that is replaced with its definition before the makefile is executed. Although the macro name can consist of lowercase or uppercase characters, uppercase is an accepted convention. The general form of a macro definition is:

```
MACRO = text
MACRO += and more text
```

Spaces around the equal sign are not significant. With the **+=** operator you can add a string to an existing macro. An extra space is inserted before the added string automatically.

To use a macro, you must access its contents:

```
$(MACRO)        # you can read this as
${MACRO}        # the contents of macro MACRO
```

If the macro name is a single character, the parentheses are optional. Note that the expansion is done recursively, so the body of a macro may contain other macros. These macros are expanded when the macro is actually used, not at the point of definition:

```
FOOD = $(EAT) and $(DRINK)
EAT = meat and/or vegetables
DRINK = water
export FOOD
```

The macro `FOOD` is expanded as `meat and/or vegetables and water` at the moment it is used in the export line, and the environment variable FOOD is set accordingly.

Predefined macros

Macro	Description
`MAKE`	Holds the value **mktc**. Any line which uses `MAKE`, temporarily overrides the option **-n** (Show commands without executing), just for the duration of the one line. This way you can test nested calls to `MAKE` with the option **-n**.
`MAKEFLAGS`	Holds the set of options provided to **mktc** (except for the options **-f** and **-d**). If this macro is exported to set the environment variable MAKEFLAGS, the set of options is processed before any command line options. You can pass this macro explicitly to nested **mktc**'s, but it is also available to these invocations as an environment variable.

Macro	Description

`PRODDIR`	Holds the name of the directory where **mktc** is installed. You can use this macro to refer to files belonging to the product, for example a library source file.
	
	`DOPRINT = $(PRODDIR)/lib/src/_doprint.c`
	
	When **mktc** is installed in the directory `c:/Tasking/bin` this line expands to:
	
	`DOPRINT = c:/Tasking/lib/src/_doprint.c`

`SHELLCMD`	Holds the default list of commands which are local to the SHELL. If a rule is an invocation of one of these commands, a SHELL is automatically spawned to handle it.
`$`	This macro translates to a dollar sign. Thus you can use "$$" in the makefile to represent a single "$".

Dynamically maintained macros

There are several dynamically maintained macros that are useful as abbreviations within rules. It is best not to define them explicitly.

Macro	Description
`$*`	The basename of the current target.
`$<`	The name of the current dependency file.
`$@`	The name of the current target.
`$?`	The names of dependents which are younger than the target.
`$!`	The names of all dependents.

The $< and $* macros are normally used for implicit rules. They may be unreliable when used within explicit target command lines. All macros may be suffixed with **F** to specify the Filename components (e.g. `${*F}`, `${@F}`). Likewise, the macros $*, $< and $@ may be suffixed by **D** to specify the Directory component.

The result of the $* macro is always without double quotes ("), regardless of the original target having double quotes (") around it or not.

The result of using the suffix **F** (Filename component) or **D** (Directory component) is also always without double quotes ("), regardless of the original contents having double quotes (") around it or not.

8.2.2.4. Makefile Functions

A function not only expands but also performs a certain operation. Functions syntactically look like macros but have embedded spaces in the macro name, e.g. '$(match arg1 arg2 arg3)'. All functions are built-in and currently these are: `match`, `separate`, `protect`, `exist`, `nexist` and `addprefix`.

$(match *suffix filename* ...)

The `match` function yields all arguments which match a certain suffix:

```
$(match .o prog.o sub.o mylib.a)
```

yields:

```
prog.o sub.o
```

$(separate *separator argument* ...)

The `separate` function concatenates its arguments using the first argument as the separator. If the first argument is enclosed in double quotes then '\n' is interpreted as a newline character, '\t' is interpreted as a tab, '*ooo*' is interpreted as an octal value (where, *ooo* is one to three octal digits), and spaces are taken literally. For example:

```
$(separate "\n" prog.o sub.o)
```

results in:

```
prog.o
sub.o
```

Function arguments may be macros or functions themselves. So,

```
$(separate "\n" $(match .o $!))
```

yields all object files the current target depends on, separated by a newline string.

$(protect *argument*)

The `protect` function adds one level of quoting. This function has one argument which can contain white space. If the argument contains any white space, single quotes, double quotes, or backslashes, it is enclosed in double quotes. In addition, any double quote or backslash is escaped with a backslash.

Example:

```
echo $(protect I'll show you the "protect" function)
```

yields:

```
echo "I'll show you the \"protect\" function"
```

$(exist *file | directory argument*)

The `exist` function expands to its second argument if the first argument is an existing file or directory.

Example:

```
$(exist test.c cctc test.c)
```

When the file `test.c` exists, it yields:

```
cctc test.c
```

When the file `test.c` does not exist nothing is expanded.

$(nexist *file|directory argument*)

The `nexist` function is the opposite of the `exist` function. It expands to its second argument if the first argument is not an existing file or directory.

Example:

```
$(nexist test.src cctc test.c)
```

$(addprefix *prefix, argument* ...)

The `addprefix` function adds a prefix to its arguments. It is used in `mktc.mk` for invocation of the control program to pass arguments directly to a tool.

Example:

```
cctc $(addprefix -Wc, -g1 -O2) test.c
```

yields:

```
cctc -Wc-g1 -Wc-O2 test.c
```

8.2.2.5. Conditional Processing

Lines containing `ifdef`, `ifndef`, `else` or `endif` are used for conditional processing of the makefile. They are used in the following way:

```
ifdef macro-name
if-lines
else
else-lines
endif
```

The *if-lines* and *else-lines* may contain any number of lines or text of any kind, even other `ifdef`, `ifndef`, `else` and `endif` lines, or no lines at all. The `else` line may be omitted, along with the *else-lines* following it.

First the *macro-name* after the `ifdef` command is checked for definition. If the macro is defined then the *if-lines* are interpreted and the *else-lines* are discarded (if present). Otherwise the *if-lines* are discarded; and if there is an `else` line, the *else-lines* are interpreted; but if there is no else line, then no lines are interpreted.

When you use the `ifndef` line instead of `ifdef`, the macro is tested for not being defined. These conditional lines can be nested up to 6 levels deep.

You can also add tests based on strings. With `ifeq` the result is true if the two strings match, with `ifneq` the result is true if the two strings do not match. They are used in the following way:

```
ifeq(string1,string2)
if-lines
else
else-lines
endif
```

8.2.2.6. Comment, Include and Export Lines

Comment lines

Anything after a "#" is considered as a comment, and is ignored. If the "#" is inside a quoted string, it is not treated as a comment. Completely blank lines are ignored.

```
test.src  : test.c       # this is comment and is
            cctc test.c  # ignored by the make utility
```

Include lines

An *include line* is used to include the text of another makefile (like including a `.h` file in a C source). Macros in the name of the included file are expanded before the file is included. You can include several files. Include files may be nested.

```
include makefile2 makefile3
```

Export lines

An *export line* is used to export a macro definition to the environment of any command executed by the make utility.

```
GREETING = Hello
export GREETING
```

This example creates the environment variable `GREETING` with the value `Hello`. The macro is exported at the moment the export line is read so the macro definition has to precede the export line.

8.3. Make Utility amk

amk is the make utility Eclipse uses to maintain, update, and reconstruct groups of programs. But you can also use it on the command line. Its features are a little different from **mktc**. The main difference compared to **mktc** and other make utilities, is that **amk** features parallelism which utilizes the multiple cores found on modern host hardware, hardening for path names with embedded white space and it has an (internal) interface to provide progress information for updating a progress bar. It does not use an external command shell (`/bin/sh`, `cmd.exe`) but executes commands directly.

The primary purpose of any make utility is to speed up the edit-build-test cycle. To avoid having to build everything from scratch even when only one source file changes, it is necessary to describe dependencies between source files and output files and the commands needed for updating the output files. This is done in a so called "makefile".

8.3.1. Makefile Rules

A makefile dependency rule is a single line of the form:

```
[target ...] : [prerequisite ...]
```

where *target* and *prerequisite* are path names to files. Example:

```
test.o : test.c
```

This states that target `test.o` depends on prerequisite `test.c`. So, whenever the latter is modified the first must be updated. Dependencies accumulate: prerequisites and targets can be mentioned in multiple dependency rules (circular dependencies are not allowed however). The command(s) for updating a target when any of its prerequisites have been modified must be specified with leading white space after any of the dependency rule(s) for the target in question. Example:

```
test.o :
  cctc test.c   # leading white space
```

Command rules may contain dependencies too. Combining the above for example yields:

```
test.o : test.c
  cctc test.c
```

White space around the colon is not required. When a path name contains special characters such as '**:**', '**#**' (start of comment), '**=**' (macro assignment) or any white space, then the path name must be enclosed in single or double quotes. Quoted strings can contain anything except the quote character itself and a newline. Two strings without white space in between are interpreted as one, so it is possible to embed single and double quotes themselves by switching the quote character.

When a target does not exist, its modification time is assumed to be very old. So, **amk** will try to make it. When a prerequisite does not exist possibly after having tried to make it, it is assumed to be very new. So, the update commands for the current target will be executed in that case. **amk** will only try to make targets which are specified on the command line. The default target is the first target in the makefile which does not start with a dot.

Static pattern rules

Static pattern rules are rules which specify multiple targets and construct the prerequisite names for each target based on the target name.

```
[target ...] : target-pattern : [prerequisite-patterns ...]
```

The *target* specifies the targets the rules applies to. The *target-pattern* and *prerequisite-patterns* specify how to compute the prerequisites of each target. Each target is matched against the *target-pattern* to extract a part of the target name, called the *stem*. This stem is substituted into each of the *prerequisite-patterns* to make the prerequisite names (one from each *prerequisite-pattern*).

Each pattern normally contains the character '%' just once. When the *target-pattern* matches a target, the '%' can match any part of the target name; this part is called the *stem*. The rest of the pattern must match exactly. For example, the target `foo.o` matches the pattern '`%.o`', with '`foo`' as the stem. The targets `foo.c` and `foo.elf` do not match that pattern.

The prerequisite names for each target are made by substituting the stem for the '%' in each prerequisite pattern.

Example:

```
objects = test.o filter.o

all: $(objects)

$(objects): %.o: %.c
    cctc -c $< -o $@
    echo the stem is $*
```

Here '`$<`' is the automatic variable that holds the name of the prerequisite, '`$@`' is the automatic variable that holds the name of the target and '`$*`' is the stem that matches the pattern. Internally this translates to the following two rules:

```
test.o: test.c
    cctc -c test.c -o test.o
    echo the stem is test

filter.o: filter.c
    cctc -c filter.c -o filter.o
    echo the stem is filter
```

Each target specified must match the target pattern; a warning is issued for each target that does not.

Special targets

There are a number of special targets. Their names begin with a period.

Target	Description
.DEFAULT	If you call the make utility with a target that has no definition in the makefile, this target is built.

Target	Description
.DONE	When the make utility has finished building the specified targets, it continues with the rules following this target.
.INIT	The rules following this target are executed before any other targets are built.

.PHONY	The prerequisites of this target are considered to be phony targets. A phony target is a target that is not really the name of a file. The rules following a phony target are executed unconditionally, regardless of whether a file with that name exists or what its last-modification time is.
	
	For example:
	
	`.PHONY: clean`
	
	`clean:`
	        `rm *.o`
	
	With `amk clean`, the command is executed regardless of whether there is a file named `clean`.

8.3.2. Makefile Directives

Directives inside makefiles are executed while reading the makefile. When a line starts with the word "`include`" or "`-include`" then the remaining arguments on that line are considered filenames whose contents are to be inserted at the current line. "`-include`" will silently skip files which are not present. You can include several files. Include files may be nested.

Example:

```
include makefile2 makefile3
```

White spaces (tabs or spaces) in front of the directive are allowed.

8.3.3. Macro Definitions

A *macro* is a symbol name that is replaced with its definition before the makefile is executed. Although the macro name can consist of lowercase or uppercase characters, uppercase is an accepted convention. When a line does not start with white space and contains the assignment operator '**=**', '**:=**' or '**+=**' then the line is interpreted as a macro definition. White space around the assignment operator and white space at the end of the line is discarded. Single character macro evaluation happens by prefixing the name with '**$**'. To evaluate macros with names longer than one character put the name between parentheses '**()**' or curly braces '**{}**'. Macro names may contain anything, even white space or other macro evaluations. Example:

```
DINNER = $(FOOD) and $(BEVERAGE)
FOOD = pizza
BEVERAGE = sparkling water
FOOD += with cheese
```

With the **+=** operator you can add a string to an existing macro. An extra space is inserted before the added string automatically.

Macros are evaluated recursively. Whenever $(DINNER) or ${DINNER} is mentioned after the above, it will be replaced by the text "pizza with cheese and sparkling water". The left hand side in a macro definition is evaluated before the definition takes place. Right hand side evaluation depends on the assignment operator:

=	Evaluate the macro at the moment it is used.
:=	Evaluate the replacement text before defining the macro.

Subsequent '**+=**' assignments will inherit the evaluation behavior from the previous assignment. If there is none, then '**+=**' is the same as '**=**'. The default value for any macro is taken from the environment. Macro definitions inside the makefile overrule environment variables. Macro definitions on the **amk** command line will be evaluated first and overrule definitions inside the makefile.

Predefined macros

Macro	Description
$	This macro translates to a dollar sign. Thus you can use "$$" in the makefile to represent a single "$".
@	The name of the current target. When a rule has multiple targets, then it is the name of the target that caused the rule commands to be run.
*	The basename (or stem) of the current target. The stem is either provided via a static pattern rule or is calculated by removing all characters found after and including the last dot in the current target name. If the target name is 'test.c' then the stem is 'test' (if the target was not created via a static pattern rule).
<	The name of the first prerequisite.
MAKE	The **amk** path name (quoted if necessary). Optionally followed by the options **-n** and **-s**.
ORIGIN	The name of the directory where **amk** is installed (quoted if necessary).
SUBDIR	The argument of option **-G**. If you have nested makes with **-G** options, the paths are combined. This macro is defined in the environment (i.e. default macro value).

The @, * and < macros may be suffixed by '**D**' to specify the directory component or by '**F**' to specify the filename component. $(@D) evaluates to the directory name holding the file$(@F). $(@D)/$(@F) is equivalent to $@. Note that on MS-Windows most programs accept forward slashes, even for UNC path names.

The result of the predefined macros @, * and < and 'D' and 'F' variants is not quoted, so it may be necessary to put quotes around it.

Note that stem calculation can cause unexpected values. For example:

```
$@                      $*
/home/.wine/test        /home/
```

```
/home/test/.project    /home/test/
/../file               /.
```

Macro string substitution

When the macro name in an evaluation is followed by a colon and equal sign as in

```
$(MACRO:string1=string2)
```

then **amk** will replace *string1* at the end of every word in $(MACRO) by *string2* during evaluation. When $(MACRO) contains quoted path names, the quote character must be mentioned in both the original string and the replacement string[1]. For example:

```
$(MACRO:.o"=.d")
```

8.3.4. Makefile Functions

A function not only expands but also performs a certain operation. The following functions are available:

$(filter *pattern ...,item ...*)

The filter function filters a list of items using a pattern. It returns *items* that do match any of the *pattern* words, removing any items that do not match. The patterns are written using '%',

```
${filter %.c %.h, test.c test.h test.o readme.txt .project output.c}
```

results in:

```
test.c test.h output.c
```

$(filter-out *pattern ...,item ...*)

The filter-out function returns all *items* that do not match any of the *pattern* words, removing the items that do match one or more. This is the exact opposite of the filter function.

```
${filter-out %.c %.h, test.c test.h test.o readme.txt .project output.c}
```

results in:

```
test.o readme.txt .project
```

$(foreach *var-name, item ..., action*)

The foreach function runs through a list of items and performs the same *action* for each *item*. The *var-name* is the name of the macro which gets dynamically filled with an item while iterating through the *item* list. In the *action* you can refer to this macro. For example:

```
${foreach T, test filter output, ${T}.c ${T}.h}
```

---

[1]Internally, **amk** tokenizes the evaluated text, but performs substitution on the original input text to preserve compatibility here with existing make implementations and POSIX.

results in:

```
test.c test.h filter.c filter.h output.c output.h
```

8.3.5. Conditional Processing

Lines containing `ifdef`, `ifndef`, `else` or `endif` are used for conditional processing of the makefile. They are used in the following way:

ifdef *macro-name*
if-lines
else
else-lines
endif

The *if-lines* and *else-lines* may contain any number of lines or text of any kind, even other `ifdef`, `ifndef`, `else` and `endif` lines, or no lines at all. The `else` line may be omitted, along with the *else-lines* following it. White spaces (tabs or spaces) in front of preprocessing directives are allowed.

First the *macro-name* after the `ifdef` command is checked for definition. If the macro is defined then the *if-lines* are interpreted and the *else-lines* are discarded (if present). Otherwise the *if-lines* are discarded; and if there is an `else` line, the *else-lines* are interpreted; but if there is no else line, then no lines are interpreted.

When you use the `ifndef` line instead of `ifdef`, the macro is tested for not being defined. These conditional lines can be nested to any level.

You can also add tests based on strings. With `ifeq` the result is true if the two strings match, with `ifneq` the result is true if the two strings do not match. They are used in the following way:

ifeq(*string1,string2* **)**
if-lines
else
else-lines
endif

8.3.6. Makefile Parsing

amk reads and interprets a makefile in the following order:

1. When the last character on a line is a backslash (****) (i.e. without trailing white space) then that line and the next line will be concatenated, removing the backslash and newline.

2. The unquoted '**#**' character indicates start of comment and may be placed anywhere on a line. It will be removed in this phase.

   ```
   # this comment line is continued\
   on the next line
   ```

3. Trailing white space is removed.

4. When a line starts with white space and it is not followed by a directive or preprocessing directive, then it is interpreted as a command for updating a target.

5. Otherwise, when a line contains the unquoted text '**=**', '**+=**' or '**:=**' operator, then it will be interpreted as a macro definition.

6. Otherwise, all macros on the line are evaluated before considering the next steps.

7. When the resulting line contains an unquoted '**:**' the line is interpreted as a dependency rule.

8. When the first token on the line is "include" or "-include" (which by now must start on the first column of the line), **amk** will execute the directive.

9. Otherwise, the line must be empty.

Macros in commands for updating a target are evaluated right before the actual execution takes place (or would take place when you use the **-n** option).

8.3.7. Makefile Command Processing

A line with leading white space (tabs or spaces) without a (preprocessing) directive is considered as a command for updating a target. When you use the option **-j** or **-J**, **amk** will execute the commands for updating different targets in parallel. In that case standard input will not be available and standard output and error output will be merged and displayed on standard output only after the commands have finished for a target.

You can precede a command by one or more of the following characters:

@   Do not show the command. By default, commands are shown prior to their output.

-            Continue upon error. This means that **amk** ignores a non-zero exit code of the command.

+   Execute the command, even when you use option **-n** (dry run).

| Execute the command on the foreground with standard input, standard output and error output available.

Built-in commands

Command	Description
true	This command does nothing. Arguments are ignored.
false	This command does nothing, except failing with exit code 1. Arguments are ignored.
echo *arg*...	Display a line of text.
exit *code*	Exit with defined code. Depending on the program arguments and/or the extra rule options '-' this will cause **amk** to exit with the provided code. Please note that 'exit 0' has currently no result.

Command	Description
`argfile` *file arg...*	Create an argument file suitable for the **--option-file** (**-f**) option of all the other tools. The first `argfile` argument is the name of the file to be created. Subsequent arguments specify the contents. An existing argument file is not modified unless necessary. So, the argument file itself can be used to create a dependency to options of the command for updating a target.
`rm` [*option*]... *file...*	Remove the specified file(s). The following options are available: **-r**, **--recursive** Remove directories and their contents recursively. **-f**, **--force** Force deletion. Ignore non-existent files, never prompt. **-i**, **--interactive** Interactive. Prompt before every removal. **-v**, **--verbose** Verbose mode. Explain what is being done. **-m** *file* Read options from *file.*. **-?**, **--help** Show usage.

8.3.8. Calling the amk Make Utility

The invocation syntax of **amk** is:

amk [*option*]... [*target*]... [*macro=def*]...

For example:

```
amk test.elf
```

target	You can specify any target that is defined in the makefile. A target can also be one of the intermediate files specified in the makefile.
macro=def	Macro definition. This definition remains fixed for the **amk** invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is not inherited by subordinate **amk**'s
option	For a complete list and description of all **amk** make utility options, see Section 10.8, *Parallel Make Utility Options*.

Exit status

The make utility returns an exit status of 1 when it halts as a result of an error. Otherwise it returns an exit status of 0.

8.4. Eclipse Console Utility

eclipsec is the console executable variant of Eclipse. Unlike the other utilities it is part of Eclipse and therefore it is present in the `eclipse` sub-directory of the product (*installation-path*`\ctc\eclipse`) instead of in the `bin` directory. You can use it to start Eclipse from a Windows Command Prompt. This section describes how to use this utility to perform the following actions:

• Build a project from the command line without starting the Eclipse IDE.

• Generate makefiles from the command line without starting the Eclipse IDE.

The general invocation syntax of **eclipsec** is:

installation-dir**ctc\eclipse\eclipsec** [*option*]...

Note that in this manual we only describe the options that are of interest to perform the desired actions.

8.4.1. Headless Build

The Eclipse workbench (IDE) is referred to as 'head'. Headless build essentially means running builds from the command line without starting the Eclipse IDE.

To perform a headless build

Use the following invocation syntax in a Windows Command Prompt:

installation-dir**ctc\eclipse\eclipsec -nosplash -data** *workspace-location*
        -application com.tasking.managedbuilder.headlessbuild
        -build {*project* | **all**}

Explanation of the options used:

-nosplash	Do not generate the Eclipse splash screen on startup.
-data *workspace-location*	The location of your workspace. If you used the default settings when installing the product, the workspace location is `C:\Users\`*user*`\workspace_ctc_`*version* (Windows 7 or higher).
-application	`com.tasking.managedbuilder.headlessbuild` is the application to use for a headless build.
-build {*project* | **all**}	Perform a headless build. When you specify a project name, Eclipse builds the active configuration of the specified project. With **all**, Eclipse builds all active configurations of all projects in the specified workspace.

For example, to build the active configuration of `myproject`, enter:

```
eclipsec -nosplash -data "C:\Users\User\workspace_ctc_vx.yrz"
        -application com.tasking.managedbuilder.headlessbuild
        -build myproject
```

Output similar to the following appears on the console:

```
**** Build of configuration Debug for project myproject ****

"C:\\Program Files\\TASKING\\TriCore vx.yrz\\ctc\\bin\\amk" -j1 all -a
Compiling cstart.c
Compiling myproject.c
Compiling sync_on_halt.c
Linking to myproject.elf

Time consumed: 4438 ms
**** End of build ****
```

8.4.2. Generating Makefiles from the Command Line

You can use **eclipsec** to generate makefiles for one or more of your projects, without starting the Eclipse IDE. This can be useful for testing and batch processing.

To generate makefiles on the command line

Use the following invocation syntax in a Windows Command Prompt:

```
installation-dir\ctc\eclipse\eclipsec -nosplash -data workspace-location
        -application com.tasking.managedbuilder.headlessbuild
        -generateMakefile {project[/configuration] | all}
```

Explanation of the options used:

-nosplash	Do not generate the Eclipse splash screen on startup.
-data *workspace-location*	The location of your workspace. If you used the default settings when installing the product, the workspace location is C:\Users*user*\workspace_ctc_*version* (Windows 7 or higher).
-application	com.tasking.managedbuilder.headlessbuild is the application to use for generating makefiles on the command line.
-generateMakefile {*project*[*/configuration*] | **all** | *.*/.**}	Generate makefile(s). When you specify a project name, Eclipse generates makefiles for all configurations of the specified project. When you also specify a configuration, Eclipse generates makefiles for the specified configuration of the project. With **all**, Eclipse generates makefiles for all active configurations of all projects in the specified workspace. With *.*/.** Eclipse generates makefiles for all configurations of all projects in the specified workspace.

For example, to generate makefiles for all configurations of `myproject`, enter:

```
eclipsec -nosplash -data "C:\Users\User\workspace_ctc_vx.yrz"
        -application com.tasking.managedbuilder.headlessbuild
        -generateMakefile myproject
```

Output similar to the following appears on the console:

```
**** Build Makefile of configuration Debug for project myproject ****


**** Build Makefile of configuration Release for project myproject ****
```

The `Debug` and `Release` directory of `myproject` now contain updated makefiles.

8.5. Archiver

The archiver **artc** is a program to build and maintain your own library files. A library file is a file with extension `.a` and contains one or more object files (`.o`) that may be used by the linker.

The archiver has five main functions:

• Deleting an object module from the library

• Moving an object module to another position in the library file

• Replacing an object module in the library or add a new object module

• Showing a table of contents of the library file

• Extracting an object module from the library

The archiver takes the following files for input and output:
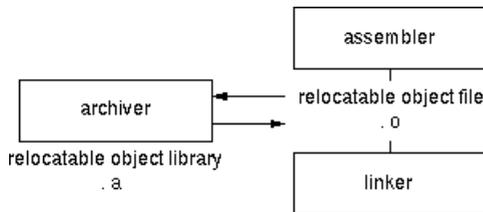
The linker optionally includes object modules from a library if that module resolves an external symbol definition in one of the modules that are read before.

8.5.1. Calling the Archiver

You can create a library in Eclipse, which calls the archiver or you can call the archiver on the command line.

To create a library in Eclipse

Instead of creating a TriCore absolute ELF file, you can choose to create a library. You do this when you create a new project with the New C/C++ Project wizard.

1. From the **File** menu, select **New » TASKING TriCore C/C++ Project**.

   The New C/C++ Project wizard appears.

2. Enter a project name.

3. In the **Project type** box, select **TASKING TriCore Library** and click**Next >**.

4. Follow the rest of the wizard and click **Finish**.

5. Add the files to your project.

6. Build the project as usual. For example, select **Project » Build Project** (🔟).

   Eclipse builds the library. Instead of calling the linker, Eclipse now calls the archiver.

Command line invocation

You can call the archiver from the command line. The invocation syntax is:

artc *key_option* [*sub_option*...] *library* [*object_file*]

key_option	With a key option you specify the main task which the archiver should perform. You must *always* specify a key option.
sub_option	Sub-options specify into more detail how the archiver should perform the task that is specified with the key option. It is not obligatory to specify sub-options.
library	The name of the library file on which the archiver performs the specified action. You must always specify a library name, except for the options **-?** and **-V**. When the library is not in the current directory, specify the complete path (either absolute or relative) to the library.
object_file	The name of an object file. You must always specify an object file name when you add, extract, replace or remove an object file from the library.

Options of the archiver utility

The following archiver options are available:

Description	Option	Sub-option
Main functions (key options)		
Replace or add an object module	**-r**	**-a -b -c -u -v**
Extract an object module from the library	**-x**	**-v**
Delete object module from library	**-d**	**-v**
Move object module to another position	**-m**	**-a -b -v**
Print a table of contents of the library	**-t**	**-s0 -s1**
Print object module to standard output	**-p**	
Sub-options		
Append or move new modules after existing module *name*	**-a** *name*	
Append or move new modules before existing module *name*	**-b** *name*	
Create library without notification if library does not exis	**-c**	
Preserve last-modified date from the library	**-o**	
Print symbols in library modules	**-s{0|1}**	
Replace only newer modules	**-u**	
Verbose	**-v**	
Miscellaneous		

Description	Option	Sub-option
Display options	**-?**	
Display version header	**-V**	
Read options from *file*	**-f** *file*	
Suppress warnings above level *n*	**-w***n*	

For a complete list and description of all archiver options, see Section 10.9, *Archiver Options*.

8.5.2. Archiver Examples

Create a new library

If you add modules to a library that does not yet exist, the library is created. To create a new library with the name `mylib.a` and add the object modules `cstart.o` and `calc.o` to it:

```
artc –r mylib.a cstart.o calc.o
```

Add a new module to an existing library

If you add a new module to an existing library, the module is added at the end of the module. (If the module already exists in the library, it is replaced.)

```
artc –r mylib.a mod3.o
```

Print a list of object modules in the library

To inspect the contents of the library:

```
artc –t mylib.a
```

The library has the following contents:

```
cstart.o
calc.o
mod3.o
```

Move an object module to another position

To move `mod3.o` to the beginning of the library, position it just before `cstart.o`:

```
artc -mb cstart.o mylib.a mod3.o
```

Delete an object module from the library

To delete the object module `cstart.o` from the library `mylib.a`:

```
artc -d mylib.a cstart.o
```

Extract all modules from the library

Extract all modules from the library `mylib.a`:

```
artc -x mylib.a
```

8.6. HLL Object Dumper

The high level language (HLL) dumper **hldumptc** is a program to dump information about an absolute object file (`.elf`). Key features are a disassembler with HLL source intermixing and HLL symbol display and a HLL symbol listing of static and global symbols.

8.6.1. Invocation

Command line invocation

You can call the HLL dumper from the command line. The invocation syntax is:

hldumptc [*option*]... *file*...

The input file must be an ELF file with or without DWARF debug info (`.elf`).

The HLL dumper can process multiple input files. Files and options can be intermixed on the command line. Options apply to all supplied files. If multiple files are supplied, the disassembly of each file is preceded by a header to indicate which file is dumped. For example:

========== file.elf ==========

For a complete list and description of all options, see Section 10.10, *HLL Object Dumper Options*. With hldumptc --help you will see the options on stdout.

8.6.2. HLL Dump Output Format

The HLL dumper produces output in text format by default, but you can also specify the XML output format with **--output-type=xml**. The XML output is mainly for use in the Eclipse editor. The output is printed on stdout, unless you specify an output file with **--output=**filename.

The parts of the output are dumped in the following order:

1. Module list

2. Section list

3. Call graph using the DWARF debug info

4. Data accesses per function

5. Section dump (disassembly)

6. HLL symbol table

7. Assembly level symbol table

8. Note sections

9. Debug control flow section

With the option **--dump-format=***flag* you can control which parts are shown. By default, all parts are shown, except parts for 3, 4 and 9.

Example

Suppose we have a simple "Hello World" program in a file called `hello.c`. We call the control program as follows:

```
cctc -g -t hello.c
```

Option **-g** tells to include DWARF debug information. Option **-t** tells to keep the intermediate files. This command results (among other files) in the file `hello.elf` (the absolute object file).

We can dump information about the ELF file with the following command:

```
hldumptc -F3 hello.elf
```

Option **-F3** enables all parts. A possible output could be (just a fraction of the actual output is shown):

```
---------- Module list ----------

Name    Full path
hello.c hello.c

---------- Section list ----------

Address   Size   Align Type    Name
80000766    20      2 text    .text.hello.main
80000008     6      1 romdata .zrodata.hello..1.str
d0000000     4      4 bss     .zdata.hello.world
800007a0    12      1 romdata .rodata.hello..2.str

---------- Call graph using the DWARF debug info ----------

+-- 0x80000766 main
    |
    +-- 0x8000077a printf
        |
        +-- 0x8000034e _doprint
            |
            +-- 0x8000051c _io_putc
            |   |
            |   +-- 0x80000746 fputc
            |       |
            |       +-- 0x80000428 _flsbuf
            |           |
            |           +-- 0x80000502 _host_write
            |           |   |
            |           |   +-- 0x800002b2 _dbg_trap_tc
            |           |       |
            |           |       +-- 0x80000220 _dbg_cacheawi
```

```
        |              |          |
        |              |          +-- 0x80000220 _dbg_cacheawi
        |              |          |
        |              |          +-- 0x80000220 _dbg_cacheawi
        |              |          |
        |              |          +-- 0x80000220 _dbg_cacheawi
        |              |          |
        |              |          +-- 0x8000000e _dbg_trap
        |              |
        |              +-- 0x800003be _fflush
        |              |   |
        |              |   +-- 0x80000502 _host_write *
        |              |   |
        |              |   +-- 0x800004e6 _host_lseek
        |              |       |
        |              |       +-- 0x800002b2 _dbg_trap_tc *
        |              |
        |              +-- 0x80000502 _host_write *
        |
        +-- 0x8000051c _io_putc *

---------- Data accesses per function ----------

No .debug_tasking_verif.rights sections present

No .debug_tasking_verif sections present

---------- Section dump ----------

        .sdecl '.zrodata.hello..1.str',DATA AT 0x80000008
        .sect  '.zrodata.hello..1.str'
        .byte 77,6f,72,6c,64,00                              ; world.

                            .sdecl '.text.hello.main', CODE AT 0x80000766
                            .sect  '.text.hello.main'
80000766 20 08       main:  sub.a       sp,#0x8
80000768 85 df 00 08        ld.a        a15,world
8000076c f4 af              st.a        [sp],a15
8000076e 91 00 00 48        movh.a      a4,#0x8000
80000772 d9 44 60 e0        lea         a4,[a4]0x7a0
80000776 1d 00 02 00        j           printf

        .sdecl '.rodata.hello..2.str',DATA AT 0x800007a0
        .sect  '.rodata.hello..2.str'
        .byte 48,65,6c,6c,6f,2c,20,25,73,21,0a,00            ; Hello, %s!..

        .sdecl '.zdata.hello.world',DATA AT 0xd0000000
        .sect  '.zdata.hello.world'
world:
        .space 4
```

```
---------- HLL symbol table ----------

Address     Size HLL Type              Name
80000766     20 void                   main()
8000077a     38 int                    printf(const char * restrict format, ...)
a0000000      4 void                   _START()
d0000000      4 char              * world [hello.c]
d0000004     20 struct                 _dbg_request [dbg.c]
d0000018     80 static char            stdin_buf[80] [_iob.c]
d0000068     80 static char            stdout_buf[80] [_iob.c]
d00000b8    200 struct _iobuf          _iob[10] [_iob.c]

---------- Assembly level symbol table ----------

Address  Size     Type Name
00000000
00000000              [.zdata.hello.world]
00000000              hello.src
80000766      20 code main
8000077a         code __main_function_end
8000077a      38 code printf
a0000000       4 code _START
d0000000       4 data world

---------- .note sections ----------
Section .note, section 53:
00000000 type: ALTIUM COMPILER NAME
0000000c name: Altium
00000014 desc: ctc

---------- Debug control flow section ----------
start offset : 0
start address: 0x80000766
code size    : 20
#entries     : 0
```

Module list

This part lists all modules (C/C++ files) found in the object file(s). It lists the filename and the complete path name at the time the module was built.

Section list

This part lists all sections found in the object file(s).

Address	The start address of the section. Hexadecimal, 8 digits, 32-bit.
Size	The size (length) of the section in bytes. Decimal, filled up with spaces.
Align	The alignment of the section in number of bytes. Decimal, filled up with spaces.

| **Type** | The section type. |
| **Name** | The name of the section. |

With option **--sections=***name*[,*name*]... you can specify a list of sections that should be dumped.

Call graph

The linker can generate a call graph in the linker map file. However, if you only have an ELF file and you need to test it, you can use the option **--dump-format=+callgraph**. You can then step through the call graph to identify the flow for debugging purposes. Some notes about the call graph:

• The call graph starts with the default entry point of the application.

• Recursive calls are marked with 'R'.

• Inline functions are marked with 'I'.

• Indirect function calls are marked with '__INDIRECT__'.

• A function is analyzed only once. When a function is called again, it is not analyzed again and this is marked with '*'.

• By default the DWARF debug information is used to generate the call graph. When no DWARF information is available the ELF information is used. Inline functions can only be detected and dumped when DWARF information is available.

• With option **--call-graph-elf-mode** you can force the call graph to use ELF symbols even when DWARF information is available. This can be useful when you want to dump information from an assembly function.

• With option **--call-graph-root=***function* you can specify the address or function name where to start the call graph (default: main()).

• When you dump the call graph for an AURIX application, by default only the shared functions are analyzed. When the call graph for the local RAM (CPU0,1,2...) are to be dumped then it is required to specify the option **--copy-table** too. The dumper uses the copy table information to dump local code.

Data accesses per function

This part appears when you specify option **--dump-format=+accesses**. This part only contains information if you have built your application with control program option **--software-partitioning-info**. The example from Section 8.10, *Integrity Check Utility* produces the following output (just a fraction of the actual output is shown):

```
Safety Class Areas:
-------------------

Address     Size   Safety
80000448     52        7
90000000     12        3
```

```
Safety Class Access Rights Table:
---------------------------------

    Target:     0    3    7
            +----------------
Source: 0   |  rwx  ---  ---
        3   |  ---  rwx  ---
        7   |  ---  r--  rwx


Data accesses:
--------------

 - Name                        value = "f1"
 - Address                     value = 80000448
 - Safety class               value = 7

   - File                      value = "safety.c"
   - Line number               value = 7
   - Referenced Entity         value = variable
   - Name                      value = "z"
   - Address                   value = 90000008
   - Safety class              value = 3
   - Indirection level         value = 255 (address is taken)
   - Action                    value = 4 (returned)

 - Name                        value = "f3"
 - Address                     value = 80000462
 - Safety class               value = 7

   - File                      value = "safety.c"
   - Line number               value = 19
   - Referenced Entity         value = variable
   - Name                      value = "y"
   - Address                   value = 90000004
   - Safety class              value = 3
   - Indirection level         value = 255 (address is taken)
   - Action                    value = 5 (passed as parameter)
   - Parameter                 value = 1
   - Direct call               value = 1
   - Name                      value = "f2"
   - Address                   value = 8000044e
   - Safety class              value = 7
```

Section dump

This part contains the disassembly. It consists of the following columns:

address column	Contains the address of the instruction or directive that is shown in the disassembly. If the section is relocatable the section start address is assumed to be 0. The address is represented in hexadecimal and has a fixed width. The address is padded with zeros. No 0x prefix is displayed. For example, on a 32-bit architecture, the address 0x32 is displayed as 00000032.
encoding column	Shows the hexadecimal encoding of the instruction (code sections) or it shows the hexadecimal representation of data (data sections). The encoding column has a maximum width of eight digits, i.e. it can represent a 32-bit hexadecimal value. The encoding is padded to the size of the data or instruction. For example, a 16-bit instruction only shows four hexadecimal digits. The encoding is aligned left and padded with spaces to fill the eight digits.
label column	Displays the label depending on the option **--symbols=[hll|asm|none]**. The default is **asm**, meaning that the low level (ELF) symbols are used. With **hll**, HLL (DWARF) symbols are used. With **none**, no symbols will be included in the disassembly.
disassembly column	For code sections the instructions are disassembled. Operands are replaced with labels, depending on the option **--symbols=[hll|asm|none]**.
	The contents of data sections are represented by directives. A new directive will be generated for each symbol. ELF labels in the section are used to determine the start of a directive. ROM sections are represented with `.byte`, `.half`, `.word` kind of directives, depending on the size of the data. RAM sections are represented with `.space` directives, with a size operand depending on the data size. This can be either the size specified in the ELF symbol, or the size up to the next label.

With option **--hex**, no directives will be generated for ROM data sections and no disassembly dump will be done for code sections. Instead a hex dump is done with the following format:

```
AAAAAAAA H0 H1 H2 H3 H4 H5 H6 H7 H8 H9 HA HB HC HD HE HF RRRRRRRRRRRRRRRR
```

where,

A = Address (8 digits, 32-bit)

Hx = Hex contents, one byte (16 bytes max)

R = ASCII representation (16 characters max)

For example:

```
                        section 7 (.rodata.hello..2.str):
800007a0 48 65 6c 6c 6f 2c 20 25 73 21 0a 00             Hello, %s!..
```

With option **--hex**, RAM sections will be represented with only a start address and a size indicator:

```
AAAAAAAA Space: 48 bytes
```

With option **--disassembly-intermix** you can intermix the disassembly with HLL source code.

HLL symbol table

This part contains a symbol listing based on the HLL (DWARF) symbols found in the object file(s). The symbols are sorted on address.

Address	The start address of the symbol. Hexadecimal, 8 digits, 32-bit.
Size	The size of the symbol from the DWARF info in bytes.
HLL Type	The HLL symbol type.
Name	The name of the HLL symbol.

HLL arrays are indicated by adding the size in square brackets to the symbol name. For example:

```
d0000018    80 static char        stdin_buf[80] [_iob.c]
```

With option **--expand-symbols=+basic-types** HLL struct and union symbols are listed including all fields. Array members are expanded in one array member per line regardless of the HLL type. For example:

```
d0000018    80 static char        stdin_buf[80] [_iob.c]
d0000018     1   char
d0000019     1   char
d000001a     1   char
  ...
d0000067     1   char
```

HLL struct and union symbols are listed by default without fields. For example:

```
d0000004    20 struct             _dbg_request [dbg.c]
```

With option **--expand-symbols** all struct, union and array fields are included as well. For the fields the types and names are indented with two spaces. For example:

```
d0000004    20 struct             _dbg_request [dbg.c]
d0000004     4   int                _errno
d0000008     1   enum               nr
d000000c    12   union              u
d000000c     4     struct             exit
d000000c     4       int                status
d000000c     8     struct             open
d000000c     4       const char       * pathname
d0000010     2       unsigned short int  flags
  ...
```

Functions are displayed with the full function prototype. Size is the size of the function. HLL Type is the return type of the function. For example:

```
8000077a    38 int                printf(const char * restrict format, ...)
```

The local and static symbols get an identification between square brackets. The filename is printed and if a function scope is known the function name is printed between the square brackets as well. If multiple files with the same name exist, the unique part of the path is added. For example:

```
80004100      4 int                 count [file.c, somefunc()]
80004104      4 int                 count [x\a.c]
80004108      4 int                 count [y\a.c, foo()]
```

Global symbols do not get information in square brackets.

Assembly level symbol table

This part contains a symbol listing based on the assembly level (ELF) symbols found in the object file(s). The symbols are sorted on address.

Address	The start address of the symbol. Hexadecimal, 8 digits, 32-bit.
Size	The size of the symbol from the ELF info in bytes. If this field is empty, the size is zero.
Type	Code or Data, depending on the section the symbol belongs to. If this field is empty, the symbol does not belong to a section.
Name	The name of the ELF symbol.

Debug control flow section

When control flow information is present in the ELF file (control program option **--control-flow-info**), this part shows information about the basic blocks and their relation.

start offset	The start seek offset in bytes from the beginning of the section.
start address	The start address of the basic block.
code size	The code size of the basic block.
#entries	The number of successor basic blocks. This value can be 0 if there are no successors.
dest. offset	The destination offset in bytes to the first, second, ... successor from the beginning of the section.

8.7. ELF Patch Utility

With the utility **elfpatch** you can change one or more section names or modify data references within a relocatable ELF object file (.o) or library (.a). This utility can be useful when you have objects and/or libraries from a third party that have to be adopted to the local configuration of an application. The input of **elfpatch** is an ELF patch command file and/or a data reference modification file and one library or object file.

The invocation syntax is:

elfpatch --command-file=`command-file` [`option`]... `ELF-file`

or:

elfpatch --data-reference-modification-file=`file` [`option`]... `ELF-file`

For a complete list and description of all options, see Section 10.11, *ELF Patch Utility Options*. With elfpatch --help you will see the options on stdout. With elfpatch --help=s you will see the syntax of the command file and the data reference modification file.

8.7.1. ELF Patch Command File

An ELF patch command file contains one or more commands to rename a section. With **rename_section** commands you can rename sections based on section name(s). Wildcards are allowed. With **rename_section_by_symbol** commands you can rename a section in which a specified symbol is defined. No wildcards are allowed. To write comments in an ELF patch command file, you can use the C++ style '//'.

To rename a section based on section name, use the following syntax:

rename_section("`section_from`**", "**`section_to`**")**

where, `section_from` and `section_to` are strings between double-quotes. Allowed characters are:

.	dot
_	underscore
0 .. 9	digits [0..9]
a .. z	lowercase characters
A .. Z	uppercase characters
*	asterisk, wildcard character, matches one or more characters Can be used more than once, but the number and order of wildcards must match in both *section_from* and *section_to*.

To rename a section based on symbol name, use the following syntax:

rename_section_by_symbol("`symbol_name`**", "**`new_section_name`**")**

where, `symbol_name` and `new_section_name` are strings between double-quotes. The section(s) where `symbol_name` occurs in will be renamed to `new_section_name`. Allowed characters are the same as above, except that wildcards (*) are not allowed.

Examples

```
// This is comment

rename_section( ".text.one", ".text.two" )
// renames section .text.one into .text.two

rename_section( ".text.*", ".text.two.*" )
// renames e.g. section .text.xyz into .text.two.xyz

rename_section( "*.something", "*.any" )
// renames e.g. section .data.something into .data.any

rename_section( ".text.*.xyz", ".text.*" )
// renames e.g. section .text.foo.xyz into .text.foo

rename_section_by_symbol( "main", ".text.progstart" )
// renames e.g. section .text.foo.main where symbol main
// resides into .text.progstart
```

Restrictions

The following restrictions apply when you use the ELF patch utility to rename sections:

• Renaming static data sections will not work when you use the `rename_section_by_symbol()` command.

   The reason for this is that the variable's name is not in the ELF symbol list (as defined by ELF). To rename a static data section you must use the `rename_section()` command. The term 'global' is used across files. A static variable is only 'global' in the sense of the file it is defined in and only for the functions following its definition.

8.7.2. Data Reference Modification File

Modification of global variables inside C functions

In an ELF patch data reference modification (DRM) file you can specify a replacement for any access to a specified global variable. This feature can be useful to ensure data consistency by replacing access to original variables by their copy. Use the following syntax:

function **:** *original_variable* **=** *new_variable***;**

where, *function* is the name of the function where the original variable appears in. *original_variable* is the name of the variable in your source. *new_variable* is the name of the variable to be used in the ELF file.

Text after '#' is considered comments. Empty lines are allowed.

By default the contents of this file is placed in a section called `.VarEd_Info`, unless you specify option **--vared-info-section=no-section**.

Example:

```
# This is comment

func : var_a = var_b;
# replace access to var_a by access to var_b
```

Modification of data references inside C structures

In an ELF patch data reference modification (DRM) file you can also specify a replacement for data references inside C structures. The data references are pointer initializers referencing global variables. The global variables can have any type. The C structures can be defined as `const`. Use the following syntax:

global_struct_variable_name **:** *original_member* **=** *new_member***;**

where, *global_struct_variable_name* is the name of the global structure where the original pointer initializer appears in. *original_member* is the name of the pointer initializer in your source. *new_member* is the name of the pointer initializer to be used in the ELF file.

All references are symbol references. Text after '#' is considered comments. Empty lines are allowed.

Example C source (`struct.c`):

```
typedef struct _X
{
    char  *p;
    short  s;
    int   *q;
} X;

char First;
char Second;
int Start;
int End;

X exam={&First,1,&Start};
```

Example DRM file (`struct-patch.drm`):

```
# This is comment

exam : First = Second;
exam : Start = End;
```

Compile the C file to create an ELF object file:

```
cctc -c struct.c
```

This results in the object file `struct.o`.

Apply the patch:

```
elfpatch -dstruct-patch.drm struct.o
```

Each occurrence of `First` in the structure `exam` will be replaced with `Second`, and the same for `Start` with `End`. This implicitly means that all occurrences in complex structures (a `struct` in a `struct`) will be replaced.

Restrictions with data reference modification

The following restrictions apply when you use the ELF patch utility with data reference modification:

• When you want to replace a `__far` variable with, for example, a `__near` variable, this is not a problem. The generated code works on any variable, with any (lower) memory qualification.

   In general: A variable defined with a memory scope `A` can be replaced by a variable defined with a smaller memory scope `B`.

   For example, `int __far X;` can be replaced by `int __near Y;` without problem.

• When you want to replace a `__near` variable with a `__far` variable, this might work. The generated code only supports variables that are in range.

   For example, `abs24` should only be replaced by `abs24`, `abs18` or less. It may happen that `abs32` will be linked and then this variable/data is in range of the instructions. If the `abs32` variable is out of range for the instruction, the linker issues an error.

• You cannot replace absolute variables (variables defined with `__at()`).

   The reason for this is that there is no relocation record for the variable, its address is used directly by the C compiler.

8.8. Expire Cache Utility

With the utility **expiretc** you can limit the size of the cache (C compiler option **--cache**) by removing all files older than a few days or by removing older files until the total size of the cache is smaller than a specified size. See also Section 11.6, *Compiler Cache*.

The invocation syntax is:

expiretc [*option*]... *cache-directory*

The compiler cache is present in the directory ctccache under the specified *cache-directory*.

For a complete list and description of all options, see Section 10.12, *Expire Cache Utility Options*. With expiretc --help you will see the options on stdout.

Examples

To remove all files older than seven days, enter:

expiretc --days=7 "*installation-dir*\mproject\.cache"

To reduce the compiler cache size to 4 MB, enter:

expiretc --megabytes=4 "*installation-dir*\mproject\.cache"

Older files are removed until the total size of the cache is smaller than 4 MB.

To clear the compiler cache, enter:

expiretc --megabytes=0 "*installation-dir*\mproject\.cache"

8.9. Proftool Utility

The utility **proftool** is used internally by the TASKING Profiler perspective of Eclipse to display profiling information.

The invocation syntax is:

proftool [*option*]... *file*

8.10. Integrity Check Utility

With the utility **ichk** you can check the integrity of an application. This utility checks access restrictions on memory in light of ASIL specifications and MPU configuration.

Based on an ISO 26262 ASIL level it is required to partition code and data into different safety groups. This is especially required within multi-core automotive applications. Different safety groups have different verification and validation criteria. Subsequently, safety groups have to be isolated from each other: bugs in low-safety-level code should not effect high-safety-level data, and high-safety-level code should not depend on low-safety-level code. So, low-safety-level code is not allowed to write in high-safety-level data sections, and high-safety-level code is not allowed to call low-safety-level functions.

The integrity check utility analyzes an ELF file for possible violations of these safety restrictions. It tries to rule out possible traps by the MPU at run-time. As this is based on local static analysis, not all possible violations can be found, and false positives may occur.

The partitioning of the application is done in LSL definitions. You can define different named safety groups by selecting either sections by name or by address ranges. The linker reads and interprets the LSL file and adds an extra section to the ELF file with all memory areas and their corresponding safety level and a list of all restrictions between different safety levels.

The compiler adds extra ELF sections with verification information about a) which variable is accessed by which function and how it is accessed and b) data flow information.

The integrity check utility reads the verification sections added by the compiler and the linker and analyzes the application for possible violations of these safety restrictions.

8.10.1. Invocation

The invocation syntax is:

ichk [*option*]... *ELF_file*

For a complete list and description of all options, see Section 10.13, *Integrity Check Utility Options*.

With `ichk --help` you will see the options on `stdout`.

The return value 0 of the utility means that no violation has been found. Otherwise the value is not 0 and an error is given.

8.10.2. Memory Access Checks

The following actions on entities can be present in C source files:

Entity action	Variable/function	Argument	Function return value
read / write / call	1.1 Access (read/write/call) of a variable/function	1.2 Access (read/write/call) of an argument	1.3 Access (read/write/call) of a function return value
pass as parameter to a function	2.1 Pass as parameter to a function (the address of) a variable/function	2.2 Pass as parameter to a function an argument	2.3 Pass as parameter to a function a function return value
return	3.1 Return (the address of) a variable/function	3.2 Return an argument	3.3 Return a function return value

Variable may represent a basic type (like int), a structure (member) or the element of an array.

For example (safety.c) (case numbers correspond to the numbers in the table above):

```
int x; // ASIL A (safety class 3)
int y; // ASIL A (safety class 3)
int z; // ASIL A (safety class 3)

int * f1(void)  // safety class 7
{
  return &z;    // case 3.1 - No safety issue.
}

int f2(int *n)  // safety class 7
{
  x = 0;        // case 1.1 - Write access for x can be checked with an LSL safety class.
  *f1() = 0;    // case 1.3 - Write access for z can be checked with an LSL safety class.
  return *n;    // case 1.2 - Read access for x, y, and z can be checked with an LSL safety class.
}

void f3(int *n) // safety class 7
{
  f2(&y);       // case 2.1 - Call access can be checked with an LSL safety class.
  f2(n);        // case 2.2 - Call access can be checked with an LSL safety class.
  f2(f1());     // case 2.3 - Call access can be checked with an LSL safety class.
}

void main(void)
{
  f3(&x);
}
```

8.10.3. Define Safety Class Areas and Access Rights

Safety class areas

In an LSL file you can define safety class areas with the LSL keyword `safety_class` in a `group` statement.

For example (`safety.lsl`):

```
#define SC_ASIL_A 3
#define SC_ASIL_D 7

section_layout :vtc:linear
{
    group ASIL_D (safety_class = SC_ASIL_D)
    {
        select ".text.safety.f*";
        // select f1(), f2(), f3()
    }
}
section_layout :vtc:abs18
{
    group ASIL_A (safety_class = SC_ASIL_A)
    {
        select ".zbss.safety.*";
        //select x, y, z
    }
}
```

The group names will appear in the linker map file.

Safety class access rights

In the `section_setup` part of an LSL file you can define the safety class access rights between safety class areas with the `section_reference_restriction` statement.

For example (`safety.lsl`):

```
// #define SC_ASIL_A 3
// #define SC_ASIL_D 7

section_setup :vtc:linear
{
    section_reference_restriction
    {
        safety_class = SC_ASIL_D;
        target_safety_class = SC_ASIL_A;
        attributes = r;
        // safety class '7' only has read access in safety class '3'
    }
}
```

8.10.4. Output of the Integrity Check Utility

This section contains an example output of the integrity check utility. As said before, the integrity check utility reads the verification sections added by the compiler and the linker and analyzes the application for possible violations of these safety restrictions. So, in order to add the verification sections you need to call the C compiler and linker with option **--software-partitioning-info** and specify the LSL file with safety groups and access rights to the linker. You can use Eclipse or the control program for that.

For example,

```
cctc -Ctc27x -t --software-partioning-info safety.c -Wl-dsafety.lsl
```

This results in the ELF file `safety.elf`. You can now check this file with **ichk**.

```
ichk case_test.elf -osafety.ichk --error-file=safety.eri
```

Errors and warnings are sent to `safety.eri`.

```
ichk E286: ["safety.c" 12] safety violation writing "x" (class 3) from "f2" (class 7)
ichk E286: ["safety.c" 13] safety violation writing "z" (class 3) from "f2" (class 7)
ichk E284: ["safety.c" 26] safety violation calling "f3" (class 7) from "main" (class 0)
3 errors, 0 warnings
```

The output of **ichk** is sent to `safety.ichk`.

```
Safety Class Areas:
-------------------


Address     Size  Safety
80000448      52       7
90000000      12       3



Safety Class Access Rights Table:
-------------------------------


    Target:    0    3    7
           +---------------
Source: 0  |  rwx  ---  ---
        3  |  ---  rwx  ---
        7  |  ---  r--  rwx



Call/Data Graph:
----------------


        +--f1() -> [&z]
        |   :
        |   :.. [act] Returns variable "z"
        |
        |   +--f1() -> [&z]
```

```
     |   |   :
     |   |   :.. [act] Returns variable "z"
     |   |
     +--f2(&y,&x,&z) -> []
     |   :
     |   :.. [act] Writes to variable "x"
     |   :.. [chk] Check access for writing "x" (90000000) with safety class 3 from "f2" (8000044e)
     |   :          with safety class 7 // ACCESS VIOLATION!
     |   :.. [act] Calls function "f1"
     |   :.. [chk] Check access for calling "f1" (80000448) with safety class 7 from "f2" (8000044e)
     |   :          with safety class 7
     |   :.. [act] Reads from parameter #1
     |   :.. [chk] Check access for reading "y" (90000004) with safety class 3 from "f2" (8000044e)
     |   :          with safety class 7
     |   :.. [chk] Check access for reading "x" (90000000) with safety class 3 from "f2" (8000044e)
     |   :          with safety class 7
     |   :.. [chk] Check access for reading "z" (90000008) with safety class 3 from "f2" (8000044e)
     |   :          with safety class 7
     |   :.. [act] Writes to the return value of function "f1"
     |   :.. [chk] Check access for writing "z" (90000008) with safety class 3 from "f2" (8000044e)
     |              with safety class 7 // ACCESS VIOLATION!
     |
   +--f3(&x)
   |   :
   |   :.. [act] Passes variable "y" as parameter #1 to function "f2"
   |   :.. [act] Calls function "f1"
   |   :.. [chk] Check access for calling "f1" (80000448) with safety class 7 from "f3" (80000462)
   |   :          with safety class 7
   |   :.. [act] Calls function "f2"
   |   :.. [chk] Check access for calling "f2" (8000044e) with safety class 7 from "f3" (80000462)
   |   :          with safety class 7
   |   :.. [act] Passes parameter #1 as parameter #1 to function "f2"
   |   :.. [act] Passes the return value of function "f1" as parameter #1 to function "f2"
   |
 *--main()
     :
     :.. [act] Passes variable "x" as parameter #1 to function "f3"
     :.. [act] Calls function "f3"
     :.. [chk] Check access for calling "f3" (80000462) with safety class 7 from "main" (8000047c)
              with safety class 0 // ACCESS VIOLATION!
```

Notes on the output:

- A '-' in the Safety Class Access Rights Table means 'no rights'.

- `[act]` in the Call/Data Graph means an action.

- `[chk]` in the Call/Data Graph means a check.

8.10.5. Restrictions

The following restrictions apply when you use the integrity check utility:

• General global data initialization is not covered. For example,

```
int *my_global_ptr = &my_global_int;
void foo( void )
{
    *my_global_ptr = 3;
}
```

• Indirect global pointers are not covered. For example,

```
int *my_global_ptr;
void foo( void )
{
    my_global_ptr = &my_global_int;
    *my_global_ptr = 3;
}
```

• Access to a structure member is interpreted as access to the whole structure.

• Direct memory access, for example an SFR or via a constant value, is not covered.

• Temporary variables can be optimized away depending on the level of optimizations.

Chapter 9. Using the Debugger

This chapter describes the debugger and how you can run and debug a C or C++ application. This chapter only describes the TASKING specific parts.

9.1. Reading the Eclipse Documentation

Before you start with this chapter, it is recommended to read the Eclipse documentation first. It provides general information about the debugging process. This chapter guides you through a number of examples using the TASKING debugger with simulation as target.

You can find the Eclipse documentation as follows:

1. Start Eclipse.

2. From the **Help** menu, select **Help Contents**.

   The help screen overlays the Eclipse Workbench.

3. In the left pane, select **C/C++ Development User Guide**.

4. Open the **Getting Started** entry and select **Debugging projects**.

   This Eclipse tutorial provides an overview of the debugging process. Be aware that the Eclipse example does not use the TASKING tools and TASKING debugger.

9.2. Creating a Customized Debug Configuration

Before you can debug a project, you need a Debug launch configuration. Such a configuration, identified by a name, contains all information about the debug project: which debugger is used, which project is used, which binary debug file is used, which perspective is used, ... and so forth.

If you want to debug on a target board, you have to create a custom debug configuration for your target board, otherwise you have to create a debug launch configuration for the TASKING simulator.

> To debug a project, you need at least one opened and active project in your workbench. In this chapter, it is assumed that the `myproject` is opened and active in your workbench.

Create or customize your debug configuration

To create or change a debug configuration follow the steps below.

1. From the **Debug** menu, select **Debug Configurations...**

   The Debug Configurations dialog appears.

2. Select **TASKING C/C++ Debugger** and click the **New launch configuration** button (

) to add a new configuration.
Or: In the left pane, select the configuration you want to change, for example, **TASKING C/C++
Debugger » myproject.simulator**.

3. In the **Name** field enter the name of the configuration. By default, this is the name of the project, but
   you can give your configuration any name you want to distinguish it from the project name. For
   example enter `myproject.simulator` to identify the simulator debug configuration.

4. On the **Target** tab, select the **TriCore 1.3 Instruction Set Simulator** or any of the target boards.

The dialog shows several tabs.

Target tab

On the **Target** tab you can select on which target the application should be debugged. An application
can run on an external evaluation board, or on a simulator using your own PC. On this tab you can also
select the connection settings (DAS, RS-232, TCP/IP). The information in this tab is based on the Debug
Target Configuration (DTC) files as explained in Chapter 17, *Debug Target Configuration Files*.
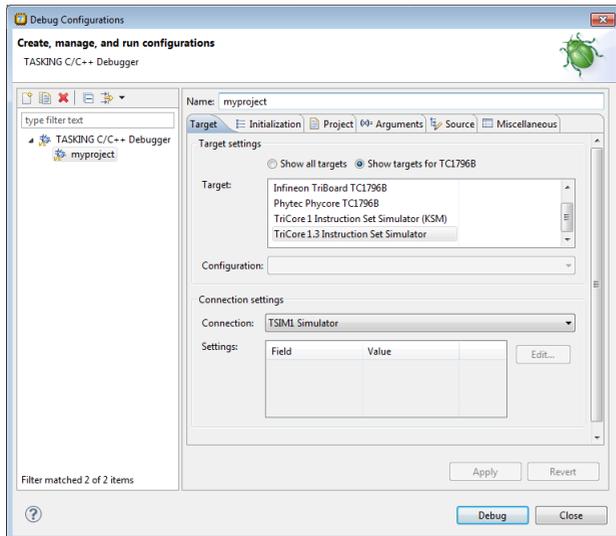
Initialization tab

On the **Initialization** tab enable one or more of the following options:
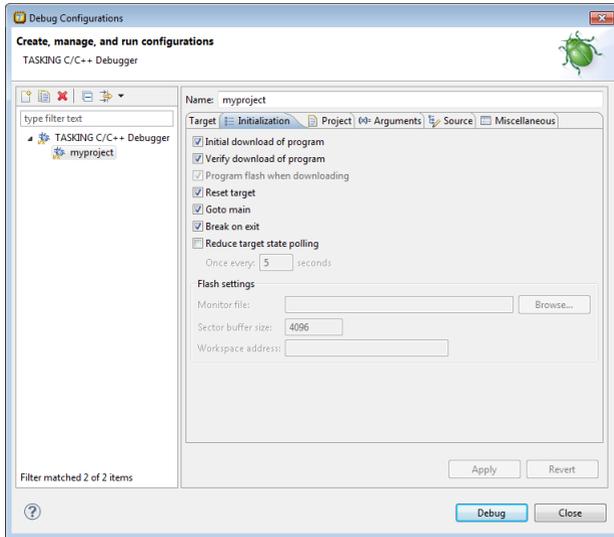
- **Initial download of program**

  If enabled, the target application is downloaded onto the target. If disabled, only the debug information in the file is loaded, which may be useful when the application has already been downloaded (or flashed) earlier. If downloading fails, the debugger will shut down.

- **Verify download of program**

  If enabled, the debugger verifies whether the code and data has been downloaded successfully. This takes some extra time but may be useful if the connection to the target is unreliable.

- **Program flash when downloading**

  If enabled, also flash devices are programmed (if necessary). Flash programming will not work when you use a simulator.

- **Reset target**

  If enabled, the target is immediately reset after downloading has completed.

- **Goto main**

  If enabled, only the C startup code is processed when the debugger is launched. The application stops executing when it reaches the first C instruction in the function `main()`. Usually you enable this option in combination with the option **Reset Target**.

- **Break on exit**

  If enabled, the target halts automatically when the `exit()` function is called.

- **Reduce target state polling**

If you have set a breakpoint, the debugger checks the status of the target every *number* of seconds to find out if the breakpoint is hit. In this field you can change the polling frequency.

- **Monitor file (Flash settings)**

  Filename of the monitor, usually an Intel Hex or S-Record file.

- **Sector buffer size (Flash settings)**

  Specifies the buffer size for buffering a flash sector.

- **Workspace address (Flash settings)**

  The address of the workspace of the flash programming monitor.

Project tab

On the **Project** tab, you can set the properties for the debug configuration such as a name for the project and the application binary file which are used when you choose this configuration.
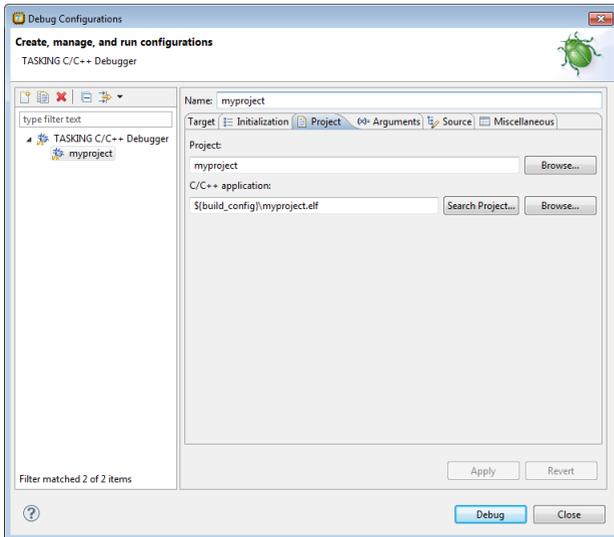
- In the **Project** field, you can choose the project for which you want to make a debug configuration. Because the project `myproject` is the active project, this project is filled in automatically. Click the Browse... button to select a different project. Only the *opened* projects in your workbench are listed.

- In the **C/C++ Application** field, you can choose the binary file to debug. The file `myproject.elf` is automatically selected from the active project.

Arguments tab

If your application's `main()` function takes arguments, you can pass them in this tab. Arguments are conventionally passed in the `argv[]` array. Because this array is allocated in target memory, make sure you have allocated sufficient memory space for it.

- In the C/C++ perspective select **Project » Properties for** to open the Properties dialog. Expand **C/C++ Build » Startup Configuration**. Enable the option **Enable passing argc/argv to main()** and specify a **Buffer size for argv**.
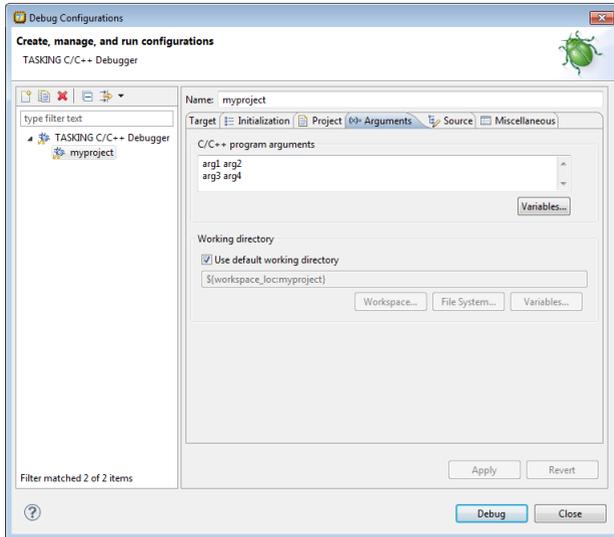
Source tab

On the **Source** tab, you can add additional source code locations in which the debugger should search for debug data.
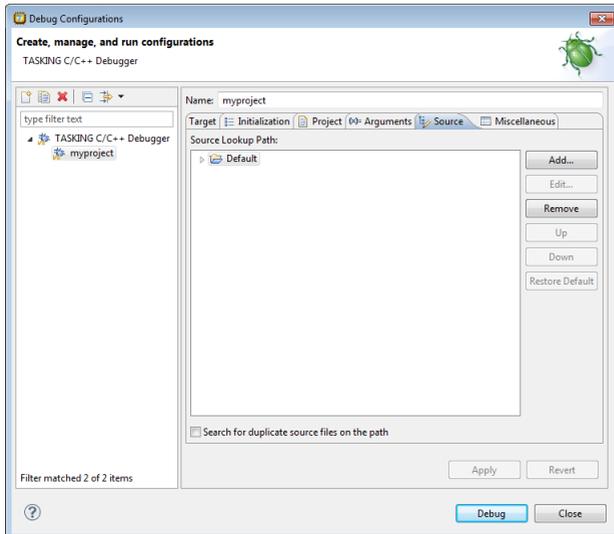
- Usually, the default source code location is correct.

Miscellaneous tab

On the **Miscellaneous** tab you can specify several file locations.
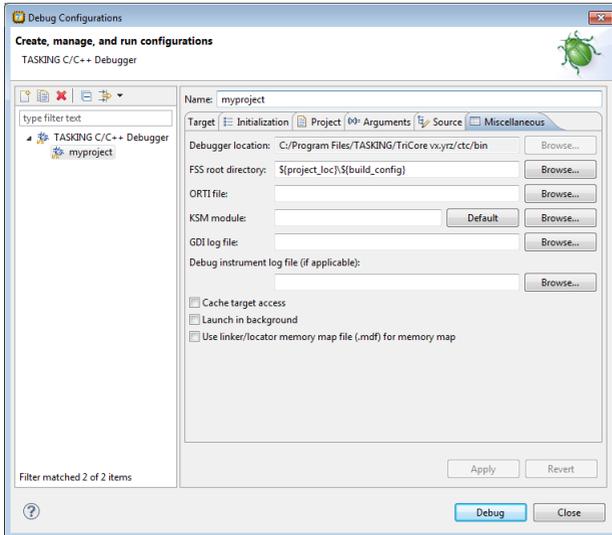
- **Debugger location**

  The location of the debugger itself. This should not be changed.

- **FSS root directory**

  The initial directory used by file system simulation (FSS) calls. See the description of the FSS view.

- **ORTI file** and **KSM module**

  If you wish to use the debugger's special facilities for kernel-aware debugging, specify the name of a Kernel Debug Interface (KDI) compatible KSM module (shared library) in the appropriate edit box. The toolset comes with a KSM suitable for RTOS kernels. If you wish to use this, browse for the file `orti_radm.dll` (Windows) or `orti_radm.so` (UNIX) in the `bin` directory of the toolset. See also the description of the RTOS view.

- **GDI log file** and **Debug instrument log file**

  You can use the options GDI log file and Debug instrument log file (if applicable) to control the generation of internal log files. These are primarily intended for use by or at the request of Altium support personnel.

- **Cache target access**

  Except when using a simulator, the debugger's performance is generally strongly dependent on the throughput and latency of the connection to the target. Depending on the situation, enabling this option may result in a noticeable improvement, as the debugger will then avoid re-reading registers and memory while the target remains halted. However, be aware that this may cause the debugger to show the wrong data if tasks with a higher priority or external sources can influence the halted target's state.

- **Launch in background**

  When this option is disabled you will see a progress bar when the debugger starts. If you do not want to see the progress bar and want that the debugger launches in the background you can enable this option.

- **Use linker/locator memory map file (.mdf) for memory map**

  You can use this option to find errors in your application that cause access to non-existent memory or cause an attempt to write to read-only memory. When building your project, the linker/locator creates a memory description file (`.mdf`) file which describes the memory regions of the target you selected in your project properties. The debugger uses this file to initialize the debugging target.

  This option is only useful in combination with a simulator as debug target. The debugger may fail to start if you use this option in combination with other debugging targets than a simulator.

9.3. Pipeline and Cache During Debugging

The pipeline and the cache(s) of the TriCore architecture are implemented in such a way that there is no automatic coherency between the state as seen by the CPU itself and that seen by the debugger via OCDS. For example, if the target halts on a breakpoint, a memory value read via OCDS may not represent the "real" value as implied by the program logic if the value still has to be written back from the cache.

The TASKING debugger has a special "sync(hronize)-on-halt" facility to bring about this coherency. Every time the target halts, the debugger will execute a routine `_sync_on_halt` that flushes the pipeline and the caches insofar as necessary. This routine is implemented in the file `ctc/lib/src/sync_on_halt.c`, which (like `cstart.c`) will be added to a new TriCore project unless you disable the option **Include debugger synchronization utility** in the New C/C++ Project wizard, which you may want to do if you do not intend to use the TASKING debugger. For example, for third-party debuggers this synchronization utility might not be necessary. In any case, by default the code will be linked in only in the Debug configuration, not in the Release configuration (via the Exclude from build facility).

The code in `sync_on_halt.c` works for most supported TriCore derivatives. For the following derivative, you must exclude the file from the build process: TC1130. When you use a user-defined processor or certain special features such as memory protection, you may have to adapt it. For all TriCore 1.3 architecture devices (such as the TC1130), only data cache flushing is supported, no instruction cache flushing. It is recommended not to switch on the program cache when debugging these devices. This is NOT the default setting for new projects.

Note that the execution of these routines at each halt may have unwanted side effects on the performance of the target application, particularly as a result of the cache flushing. The CCNT register, for instance, is also updated after each (hidden) execution of the sync on halt routine. So, when you are single stepping, the value of the CCNT register can increase rapidly. In certain cases, you may therefore want to switch off this feature, but realize that this could have a severe impact on the debugging experience. In particular, software breakpoints and File System Simulation may not work properly anymore.

9.4. Troubleshooting

If the debugger does not launch properly, this is likely due to mistakes in the settings of the execution environment or to an improper connection between the host computer and the execution environment. Always read the notes for your particular execution environment.

Some common problems you may check for, are:

Problem	Solution
Wrong device name in the launch configuration	Make sure the specified device name is correct.
Invalid baud rate	Specify baud rate that matches the baud rate the execution environment is configured to expect.
No power to the execution environment.	Make sure the execution environment or attached probe is powered.
Wrong type of RS–232 cable.	Make sure you are using the correct type of RS-232 cable.
Cable connected to the wrong port on the execution environment or host.	Some target machines and hosts have several ports. Make sure you connect the cable to the correct port.
Conflict between communication ports.	A device driver or background application may use the same communications port on the host system as the debugger. Disable any service that uses the same port-number or choose a different port-number if possible.
Port already in use by another user.	The port may already be in use by another user on some UNIX hosts, or being allocated by a login process. Some target machines and hosts have several ports. Make sure you connect the cable to the correct port.

If the program state shown by the debugger appears to deviate from the true state, check that the option **Include debugger synchronization utility** in the New C/C++ project wizard is enabled. See *Section 9.3, Pipeline and Cache During Debugging*.

9.5. TASKING Debug Perspective

After you have launched the debugger, you are either asked if the TASKING Debug perspective should be opened or it is opened automatically. The Debug perspective consists of several views.

To open views in the Debug perspective:

1. Make sure the Debug perspective is opened

2. From the **Window** menu, select **Show View »**

3. Select a view from the menu or choose **Other...** for more views.

By default, the Debug perspective is opened with the following views:
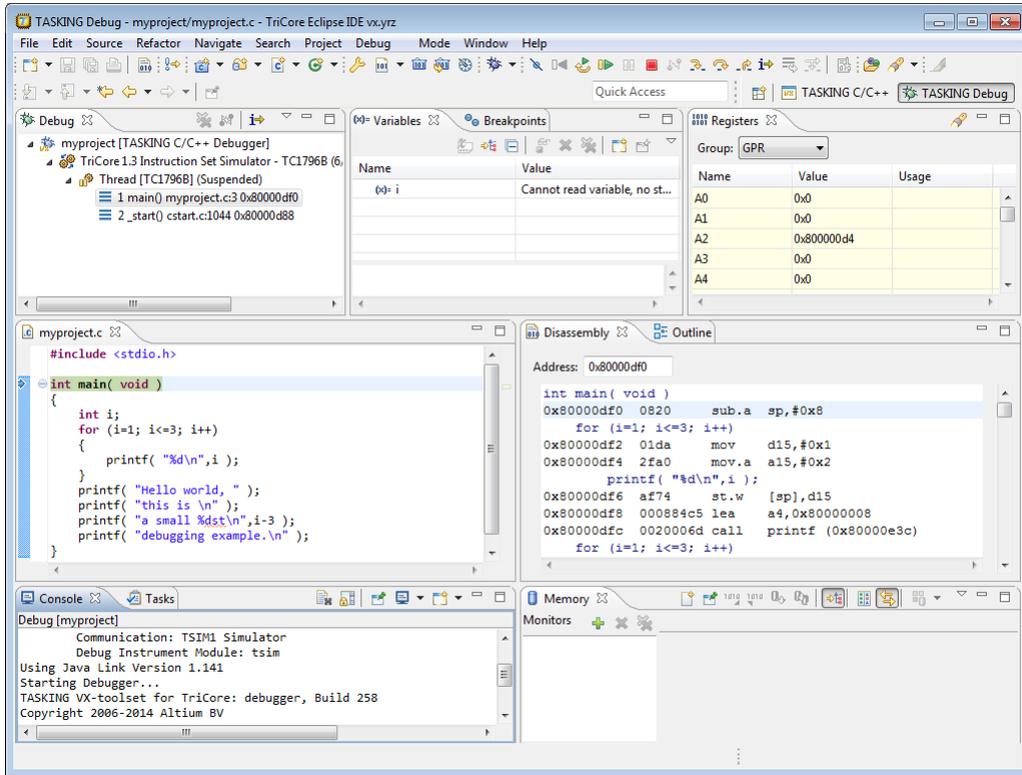
9.5.1. Debug View

The Debug view shows the target information in a tree hierarchy shown below with a sample of the possible icons:

Icon	Session item	Description
	Launch instance	Launch configuration name and launch type
	Debugger instance	Debugger name and state
	Thread instance	Thread number and state
	Stack frame instance	Stack frame number, function, file name, and file line number

Stack display

During debugging (running) the actual stack is displayed as it increases or decreases during program execution. By default, all views present information that is related to the current stack item (variables, memory, source code etc.). To obtain the information from other stack items, click on the item you want.

The Debug view displays stack frames as child elements. It displays the reason for the suspension beside the thread, (such as end of stepping range, breakpoint hit, and signal received). When a program exits, the exit code is displayed.

The Debug view contains numerous functions for controlling the individual stepping of your programs and controlling the debug session. You can perform actions such as terminating the session and stopping the program. All functions are available from the right-click menu, though commonly used functions are also available from the toolbar.

Controlling debug sessions

Icon	Action	Description
	Remove all	Removes all terminated launches.
	Reset target system	Resets the target system and restarts the application.
	Restart	Restarts the application. The target system is *not* reset.
	Resume	Resumes the application after it was suspended (manually, breakpoint, signal).
	Suspend	Suspends the application (pause). Use the Resume button to continue.
	Relaunch	Right-click menu. Restarts the selected debug session when it was terminated. If the debug session is still running, a new debug session is launched.
	Reload current application	Reloads the current application without restarting the debug session. The application does restart of course.
	Terminate	Ends the selected debug session and/or process. Use Relaunch to restart this debug session, or start another debug session.
	Terminate all	Right-click menu. As terminate. Ends *all* debug sessions.
	Terminate and remove	Right-click menu. Ends the debug session and removes it from the Debug view.
	Terminate and Relaunch	Right-click menu. Ends the debug session and relaunches it. This is the same as choosing Terminate and then Relaunch.
	Disconnect	Detaches the debugger from the selected process (useful for debugging attached processes).

Stepping through the application

Icon	Action	Description
	Step into	Steps to the next source line or instruction.
	Step over	Steps over a called function. The function is executed and the application suspends at the next instruction after the call.
	Step return	Executes the current function. The application suspends at the next instruction after the return of the function.

Icon	Action	Description
i⇥	Instruction stepping	Toggle. If enabled, the stepping functions are performed on instruction level instead of on C source line level.
⍟⇥	Interrupt aware stepping	Toggle. If enabled, the stepping functions do not step into an interrupt when it occurs.

Miscellaneous

Icon	Action	Description
📄	Copy Stack	Right-click menu. Copies the stack as text to the windows clipboard. You can paste the copied selection as text in, for example, a text editor.
🐞	Edit *project...*	Right-click menu. Opens the debug configuration dialog to let you edit the current debug configuration.
📝	Edit Source Lookup...	Right-click menu. Opens the Edit Source Lookup Path window to let you edit the search path for locating source files.

9.5.2. Breakpoints View

You can add, disable and remove breakpoints by clicking in the marker bar (left margin) of the Editor view. This is explained in the Getting Started manual.

Description

The Breakpoints view shows a list of breakpoints that are currently set. The button bar in the Breakpoints view gives access to several common functions. The right-most button ▽ opens the Breakpoints menu.

Types of breakpoints

To access the breakpoints dialog, add a breakpoint as follows:

1. Click the **Add TASKING Breakpoint** button (⚬).

   The Breakpoints dialog appears.

Each tab lets you set a breakpoint of a special type. You can set the following types of breakpoints:

* **File breakpoint**

  The target halts when it reaches the specified line of the specified source file. Note that it is possible that a source line corresponds to multiple addresses, for example when a header file has been included into two different source files or when inlining has occurred. If so, the breakpoint will be associated with all those addresses.

* **Function**

  The target halts when it reaches the first line of the specified function. If no source file has been specified and there are multiple functions with the given name, the target halts on all of those. Note that function breakpoints generally will not work on inlined instances of a function.

- **Address**

  The target halts when it reaches the specified instruction address.

- **Stack**

  The target halts when it reaches the specified stack level.

- **Data**

  The target halts when the given variable or memory location (specified in terms of an absolute address) is read or written to, as specified.

- **Instruction**

  The target halts when the given number of instructions has been executed.

- **Cycle**

  The target halts when the given number of clock cycles has elapsed.

- **Timer**

  The target halts when the given amount of time elapsed.

In addition to the type of the breakpoint, you can specify the condition that must be met to halt the program.

In the **Condition** field, type a condition. The condition is an expression which evaluates to 'true' (non-zero) or 'false' (zero). The program only halts on the breakpoint if the condition evaluates to 'true'.

In the **Ignore count** field, you can specify the number of times the breakpoint is ignored before the program halts. For example, if you want the program to halt only in the fifth iteration of a while-loop, type '4': the first four iterations are ignored.

9.5.3. File System Simulation (FSS) View

Description

The File System Simulation (FSS) view is automatically opened when the target requests FSS input or generates FSS output. The virtual terminal that the FSS view represents, follows the VT100 standard. If you right-click in the view area of the FSS view, a menu is presented which gives access to some self-explanatory functions.

VT100 characteristics

The `queens` example demonstrates some of the VT100 features. (You can find the `queens` example in the `<TriCore installation path>\examples` directory from where you can import it into your workspace.) Per debugging session, you can have more than one FSS view, each of which is associated with a positive integer. By default, the view "FSS #1" is associated with the standard streams `stdin`, `stdout`, `stderr` and `stdaux`. Other views can be accessed by opening a file named "terminal window <number>", as shown in the example below.

```
FILE * f3 = fopen("terminal window 3", "rw");
fprintf(f3, "Hello, window 3.\n");
fclose(f3);
```

You can set the initial working directory of the target application in the Debug configuration dialog (see also Section 9.2, *Creating a Customized Debug Configuration*):

1. On the **Debugger** tab, select the **Miscellaneous** sub-tab.

2. In the **FSS root directory** field, specify the FSS root directory.

The FSS implementation is designed to work without user intervention. Nevertheless, there are some aspects that you need to be aware of.

First, the interaction between the C library code (in the files dbg*.c and dbg*.h; see Section 13.1.5, *dbg.h*) and the debugger takes place via a breakpoint, which incidentally is not shown in the Breakpoints view. Depending on the situation this may be a hardware breakpoint, which may be in short supply.

Secondly, proper operation requires certain code in the C library to have debug information. This debug information should normally be present but might get lost when this information is stripped later in the development process.

When you use MIL linking/splitting the C library is translated along with your application. Therefore you need to build your application with debug information generation enabled when FSS support is needed.

9.5.4. Disassembly View

The Disassembly view shows target memory disassembled into instructions and / or data. If possible, the associated C / C++ source code is shown as well. The **Address** field shows the address of the current selected line of code.

To view the contents of a specific memory location, type the address in the **Address** field. If the address is invalid, the field turns red.

9.5.5. Expressions View

The Expressions view allows you to evaluate and watch regular C expressions.

To add an expression:

Click **OK** to add the expression.

1. Right-click in the Expressions View and select **Add Watch Expression**.

   The Add Watch Expression dialog appears.

2. Enter an expression you want to watch during debugging, for example, the variable name "i"

If you have added one or more expressions to watch, the right-click menu provides options to **Remove** and **Edit** or **Enable** and **Disable** added expressions.

- You can access target registers directly using `#NAME`. For example "`arr[#R0 << 3]`" or "`#TIMER3 = m++`". If a register is memory-mapped, you can also take its address, for example, "`&#ADCIN`".

- Expressions may contain target function calls like for example "`g1 + invert(&g2)`". Be aware that this will not work if the compiler has optimized the code in such a way that the original function code does not actually exist anymore. This may be the case, for example, as a result of inlining. Also, be aware that the function and its callees use the same stack(s) as your application, which may cause problems if there is too little stack space. Finally, any breakpoints present affect the invoked code in the normal way.

9.5.6. Memory View

Use the Memory view to inspect and change process memory. The Memory view supports the same addressing as the C and C++ languages. You can address memory using expressions such as:

- `0x0847d3c`

- `(&y)+1024`

- `*ptr`

Monitors

To monitor process memory, you need to add a *monitor*:

1. In the Debug view, select a debug session. Selecting a thread or stack frame automatically selects the associated session.

2. Click the **Add Memory Monitor** button in the Memory Monitors pane.

    The Monitor Memory dialog appears.

3. Type the address or expression that specifies the memory section you want to monitor and click **OK**.

    The monitor appears in the monitor list and the Memory Renderings pane displays the contents of memory locations beginning at the specified address.

To remove a monitor:

1. In the Monitors pane, right-click on a monitor.

2. From the popup menu, select **Remove Memory Monitor**.

Renderings

You can inspect the memory in so-called *renderings*. A rendering specifies how the output is displayed: hexadecimal, ASCII, signed integer, unsigned integer or traditional. You can add or remove renderings per monitor. Though you cannot change a rendering, you can add or remove them:

1. Click the **New Renderings...** tab in the Memory Renderings pane.

    The Add Memory Rendering dialog appears.

2. Select the rendering you want (**Traditional**, **Hex**, **ASCII**, **Signed Integer**, **Unsigned Integer** or **Hex Integer**) and click **Add Rendering(s)**.

To remove a rendering:

1. Right-click on a memory address in the rendering.

2. From the popup menu, select **Remove Rendering**.

Changing memory contents

In a rendering you can change the memory contents. Simply type a new value.

> **Warning**: Changing process memory can cause a program to crash.

The right-click popup menu gives some more options for changing the memory contents or to change the layout of the memory representation.

9.5.7. Compare Application View

You can use the Compare Application view to check if the downloaded application matches the application in memory. Differences may occur, for example, if you changed memory addresses in the Memory view.

• To check for differences, click the **Compare** button.

9.5.8. Heap View

With the Heap view you can inspect the status of the heap memory. This can be illustrated with the following example:

```
string = (char *) malloc(100);
strcpy ( string, "abcdefgh" );
free (string);
```

If you step through these lines during debugging, the Heap view shows the situation after each line has been executed. Before any of these lines has been executed, there is no memory allocated and the Heap view is empty.

• After the first line the Heap view shows that memory is occupied, the description tells where the block starts, how large it is (`100 MAUs`) and what its content is (`0x0, 0x0, ...`).

• After the second line, "`abcdefgh`" has been copied to the allocated block of memory. The description field of the Heap view again shows the actual contents of the memory block (`0x61, 0x62,...`).

• The third line frees the memory. The Heap view is empty again because after this line no memory is allocated anymore.

9.5.9. Logging View

Use the Logging view to control the generation of internal log files. This view is intended mainly for use by or at the request of Altium support personnel.

9.5.10. RTOS View

The debugger has special support for debugging real-time operating systems (RTOSs). This support is implemented in an RTOS-specific shared library called a *kernel support module* (KSM) or *RTOS-aware debugging module* (RADM). Specifically, the TASKING VX-toolset for TriCore ships with a KSM supporting the ISO 17356 standard. You have to create your own Run Time Interface (ORTI) and specify this file on the **Miscellaneous** tab while configuring a customized debug configuration (see also Section 9.2, *Creating a Customized Debug Configuration*):

1. From the **Debug** menu, select **Debug Configurations...**

   The Debug Configurations dialog appears.

2. In the left pane, select the configuration you want to change, for example, **TASKING C/C++ Debugger » myproject.simulator**.

   Or: click the **New launch configuration** button ( ) to add a new configuration.

3. Open the **Miscellaneous** tab

4. In the **ORTI file** field, specify the name of your own ORTI file.

5. If you want to use the supplied KSM suitable for RTOS kernels, in the **KSM module** field browse for the file `orti_radm.dll` (Windows) or `orti_radm.so` (UNIX) in the `ctc\bin` directory of the toolset.

The debugger supports ORTI specifications v2.0 and v2.1.

9.5.11. Registers View

In the Registers view you can examine the value of registers while stepping through your application. The registers are organized in a number of *register groups*, which together contain all known registers. You can select a group to see which registers it contains. This view has a number of features:

- While you step through the application, the registers involved in the step turn yellow. If you scroll in the view or switch groups, some registers may appear on a lighter yellow background, indicating that the debugger does not know whether the registers have changed because the debugger did not read the registers before the step began.
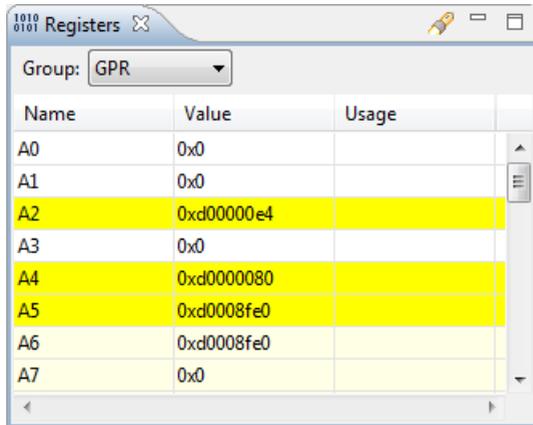
- You can change each register's value.

- For some registers the menu entry **Symbolic Representation** is available in their right-click popup menu. This opens a new view which shows the internal fields of the register. (Alternatively, you can double-click on a register). For example, the SBCU_CON register from the SBCU group may be shown as follows:
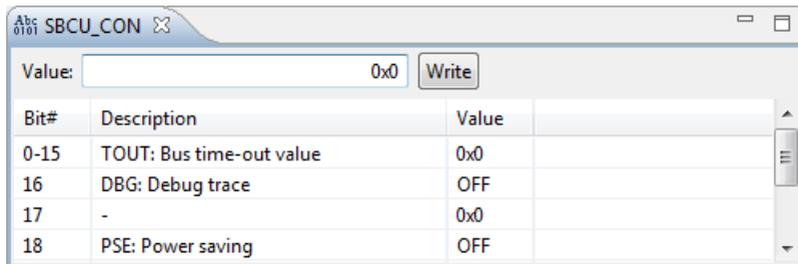
In this view you can set the individual values in the register, either by selecting a value from a drop-down box or by simply entering a value depending on the chosen field. To update the register with the new values, click the **Write** button.

- You can search for a specific register: right-click on a register and from the popup menu select **Find Register...**. Enter a group or register name filter, click the register you want to see and click **OK**. The register of your interest will be shown in the view.

9.5.12. Trace View

If tracing is enabled, the Trace view shows the code was most recently executed. For example, while you step through the application, the Trace view shows the executed code of each step. To enable tracing:

- Right-click in the Trace view and select **Trace**.

  A check mark appears when tracing is enabled.

The view has three tabs, **Source**, **Instruction** and **Raw**, each of which represents the trace in a different way. However, not all target environments will support all three of these. The view is updated automatically each time the target halts.

9.6. Multi-core Hardware Debugging

The TASKING debugger supports multi-core hardware debugging. When you start the debugger for a multi-core device (AURIX), you can debug all TriCore cores of the selected device.

The following picture shows the example `tc_queens`, which has been adapted for multi-core and is debugged on an Infineon TriBoard TC29x.
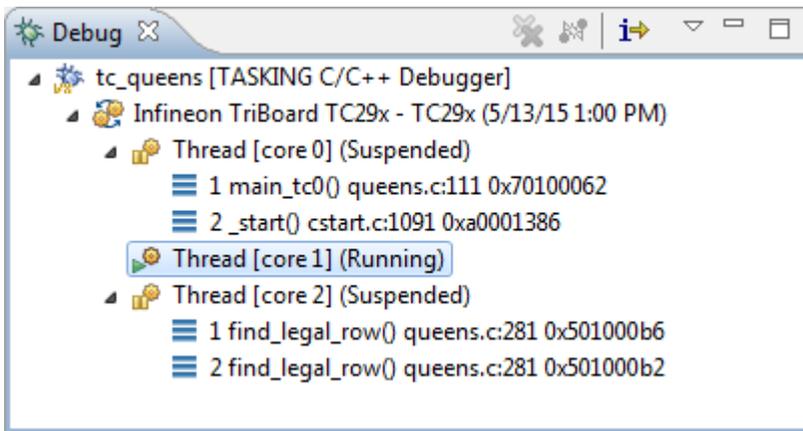
For each of the three TriCore cores of the TC29x a separate thread is started. The example shows the situation where `Thread [core 0]` is suspended, `Thread [core 1]` is running and `Thread [core 2]` is suspended.

Suspend or resume a thread/core

You can suspend or resume each of the threads independent from the other threads.

1. Select the thread you want to suspend or resume.

2. Click the **Suspend** ( ) or **Resume** ( ) button.

In the example above `Thread [core 1]` has been selected (highlighted color). When you click the **Suspend** button or **Resume** button this only effects `Thread [core 1]`.

Update views

When you select a function in a thread, the Source view, the Register view and the Disassembly view are updated to the contents of the thread (either running or suspended - with the latter situation up-to-date information is shown).

Suspend or resume all threads/cores simultaneously

When you select the board, in the picture above `Infineon TriBoard TC29x - TC29x`, you can suspend or resume all cores simultaneously. However, when one of the threads is running and the others are suspended, you must suspend the running thread before you can resume all threads at once via the board selection method. The same applies for suspending all threads at once.

Multi-core Mandelbrot example

The TriCore example project `tc_mandelbrot` demonstrates a multi-core Mandelbrot on TriBoard TC27x. You can use this example as a starting point when programming a multi-core application. See the `readme.txt` for more information about the example.

9.7. Programming a Flash Device

With the TASKING debugger you can download an application file to flash memory. Before you download the file, you must specify the type of flash devices you use in your system and the address range(s) used by these devices.

To program a flash device the debugger needs to download a flash programming monitor to the target to execute the flash programming algorithm (target-target communication). This method uses temporary target memory to store the flash programming monitor and you have to specify a temporary data workspace for interaction between the debugger and the flash programming monitor.

Two types of flash devices can exist: on-chip flash devices and external flash devices.

Setup an on-chip flash device

When you specify a target configuration board using the New C/C++ wizard or the Import Board Configuration wizard, as explained in the *Getting Started with the TASKING VX-toolset for TriCore* manual, any on-chip flash devices are setup automatically.

Setup an external flash device

1. From the **Project** menu, select **Properties for**

    The Properties for project dialog appears.

2. In the left pane, expand **Run/Debug Settings** and select **Flash Programming**.

    The Flash Programming pane appears.

3. Click **Add...** to specify an external flash device.

    The Select a New Flash Device dialog appears.

4. In the **Device type** box, expand the name of the manufacturer of the device and select a device.

    The Sector map displays the memory layout of the flash device(s). Each sector has a size and

5. In the **Base address** field enter the start address of the memory range that will be covered by the flash device. Any following addresses separated by commas are considered mirror addresses. This allows the flash device to be programmed through its mirror address before switching the flash to its base address.

6. In the **Chip width** field select the width of the flash device.

7. In the **Number of chips** field, enter the number of flash devices that are located in parallel. For example, if you have two 8-bit devices in parallel attached to a 16-bit data bus, enter 2.

8. Fill in the **Number of unused address lines** field, if necessary.

The flash memory is added to the linker script file automatically with the tag "`flash=`*flash-id*".

To program a flash device

1. From the **Debug** menu, select **Debug Configurations...**

    The Debug Configurations dialog appears.

2. In the left pane, select the configuration you want to change, for example, **TASKING C/C++ Debugger » myproject.board**.

3. On the **Debugger** tab, select the **Initialization** tab

4. Enable the option **Program flash when downloading**.

    The Flash settings group box becomes active.

5. In the **Monitor file** field, specify the filename of the flash programming monitor, usually an Intel Hex or S-Record file.

6. In the **Sector buffer size** field, specify the buffer size for buffering a flash sector.

7. Specify the data **Workspace address** used by the flash programming monitor. This address may not conflict with the addresses of the flash devices.

8. Click **Debug** to program the flash device and start debugging.

9.7.1. Boot Mode Headers

Newer TriCore devices have (typically four) Boot Mode Headers (BMHs), which lie in flash memory. An individual BMH can be either valid or invalid. For certain devices, if all BMHs are invalid, the device is normally inaccessible to the debugger. This more or less "bricks" the device because reprogramming the flash to revalidate one of the BMHs requires the debugger, or a similar program, to use the now inaccessible debug port. Recovery is possible via, for example, CAN, but this is cumbersome.

Therefore, the debugger has a special functionality to prevent all BMHs from being invalidated. The debugger only allows downloading if the target application adheres to one of the following restrictions.

- It contains at least one valid *non-ranged* BMH. "non-ranged" means that fields `ChkStart` and `ChkEnd` must be identical. This is a restriction needed for implementation reasons. You may use ranged BMHs, but they do not count as valid in this context.

- It contains no code or data in at least one of the 32-byte areas covered by a BMH. If necessary, you can instruct the linker to do this by explicitly reserving one of these ranges. The default LSL files provided with the product already do this.

Remarks

- If your application does not contain any valid BMHs itself and would overwrite the last currently valid one, the debugger will silently validate one of the other BMHs. This means that the debugger may program more flash sectors than you might expect.

- Of course, your target application has the ability to reprogram flash memory for its own purpose. This too can cause bricking, but obviously the debugger cannot prevent this.

Chapter 10. Tool Options

This chapter provides a detailed description of the options for the C compiler, C++ compiler, assembler, linker, control program, make utility, archiver, HLL object dumper and the ELF patch utility.

Tool options in Eclipse (Menu entry)

For each tool option that you can set from within Eclipse, a **Menu entry** description is available. In Eclipse you can customize the tools and tool options in the following dialog:

1. From the **Project** menu, select **Properties**

   The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   In the right pane the Settings appear.

3. Open the **Tool Settings** tab.

   You can set all tool options here.

Unless stated otherwise, all **Menu entry** descriptions expect that you have this Tool Settings tab open.

The following tables give an overview of all tool options on the Tool Settings tab in Eclipse with hyperlinks to the corresponding command line options (if available).

Global Options

Eclipse option	Description or option
Use global 'product directory' preference	Directory where the TASKING toolset is installed
Treat warnings as errors	Control program option **--warnings-as-errors**
Keep temporary files	Control program option **--keep-temporary-files (-t)**
Verbose mode of control program	Control program option **--verbose (-v)**

C/C++ Compiler

Eclipse option	Description or option
Preprocessing	
Automatic inclusion of '.sfr' file	C compiler option **--include-file**
Store preprocessor output in <file>.pre	Control program option **--preprocess (-E)** / **--no-preprocessing-only**

Eclipse option	Description or option
Keep comments in preprocessor output	Control program option **--preprocess=+comments**
Keep #line info in preprocessor output	Control program option **--preprocess=-noline**
Defined symbols	C compiler option **--define**
Pre-include files	C compiler option **--include-file**
Include Paths	
Include paths	C compiler option **--include-directory**
Precompiled C++ Headers	
Automatically use/create precompiled header file	C++ compiler option **--pch**
Create precompiled header file	C++ compiler option **--create-pch**
Use precompiled header file	C++ compiler option **--use-pch**
Precompiled header file directory	C++ compiler option **--pch-dir**
Language	
Comply to C standard	C compiler option **--iso**
Allow GNU C extensions	C compiler option **--language=+gcc**
Allow // comments in ISO C90 mode	C compiler option **--language=+comments**
Check assignment of string literal to non-const string pointer	C compiler option **--language=-strings**
Treat 'char' variables as unsigned	C compiler option **--uchar**
Treat 'int' bit-fields as signed	C compiler option **--signed-bitfields**
Always use 32-bit integers for enumeration	C compiler option **--integer-enumeration**
Allow optimization across volatile access	C compiler option **--language=-volatile**
Allow Shift JIS Kanji in strings	C compiler option **--language=+kanji**
Comply to embedded C++ subset	C++ compiler option **--embedded-c++**
Support for C++ I/O streams	C++ compiler option **--io-streams**
Support for C++ exception handling	C++ compiler option **--exceptions**
Support for C++ RTTI (run-time type information)	C++ compiler option **--rtti**
Allow the 'wchar_t' keyword (C++)	C++ compiler option **--wchar_t-keyword**
Allow non-ANSI/ISO C++ features	C++ compiler option **--strict**
C++ anachronisms	C++ compiler option **--anachronisms**
Allow GNU C++ extensions	C++ compiler option **--g++**
Floating-Point	
Floating-point model	Control program option **--fp-model**
Code Generation	
Algorithm for switch statements	C compiler option **--switch**

Eclipse option	Description or option
Minimum alignment	C compiler option **--align**
Use the MMU if present	C compiler option **--mmu-on**
Call functions indirectly	C compiler option **--indirect**
Call run-time functions indirectly	C compiler option **--indirect-runtime**
Allocation	
Clear uninitialized global and static variables	C compiler option **--no-clear**
Threshold for putting data in __near	C compiler option **--default-near-size**
Threshold for putting data in __a0	C compiler option **--default-a0-size**
Threshold for putting data in __a1	C compiler option **--default-a1-size**
Code core association	C compiler option **--code-core-association**
Data core association	C compiler option **--data-core-association**
Optimization	
Optimization level	C compiler option **--optimize**
Trade-off between speed and size	C compiler option **--tradeoff**
Maximum size for code compaction	C compiler option **--compact-max-size**
Maximum call depth for code compaction	C compiler option **--max-call-depth**
Always inline function calls	C compiler option **--inline**
Maximum size increment when inlining (in %)	C compiler option **--inline-max-incr**
Maximum size for functions to always inline	C compiler option **--inline-max-size**
Build for application wide optimizations (MIL linking)	Control program option **--mil-link** / **--mil-split**
Application wide optimization mode	Control program option **--mil-link** / **--mil-split**
Custom Optimization	C compiler option **--optimize**
Compilation Speed	C compiler option **--cache**
Debugging	
Generate symbolic debug information	C compiler option **--debug-info**
Static profiling	C compiler option **--profile=+static**
Generate profiling information for block counters	C compiler option **--profile=+block**
Generate profiling information to build a call graph	C compiler option **--profile=+callgraph**
Generate profiling information for function counters	C compiler option **--profile=+function**
Generate profiling information for function timers	C compiler option **--profile=+time**
Exclude time spent in interrupt functions	C compiler option **--profile=+time,+interrupt**
Generate code for bounds checking	C compiler option **--runtime=+bounds**
Generate code to detect unhandled case in a switch	C compiler option **--runtime=+case**
Generate code for malloc consistency checks	C compiler option **--runtime=+malloc**
MISRA C	

Eclipse option	Description or option
MISRA C checking	C compiler option **--misrac**
MISRA C version	C compiler option **--misrac-version**
Warnings instead of errors for mandatory rules	C compiler option **--misrac-mandatory-warnings**
Warnings instead of errors for required rules	C compiler option **--misrac-required-warnings**
Warnings instead of errors for advisory rules	C compiler option **--misrac-advisory-warnings**
Custom 1998 / Custom 2004 / Custom 2012	C compiler option **--misrac**
CERT C Secure Coding	
CERT C secure code checking	C compiler option **--cert**
Warnings instead of errors	C compiler option **--warnings-as-errors**
Custom CERT C	C compiler option **--cert**
Diagnostics	
Suppress C compiler warnings	C compiler option **--no-warnings=***num*
Suppress all warnings	C compiler option **--no-warnings**
Suppress C++ compiler "used before set" warnings	C++ compiler option **--no-use-before-set-warnings**
Issue remarks on C++ code	C++ compiler option **--remarks**
Perform global type checking on C code	C compiler option **--global-type-checking**
Miscellaneous	
Merge C source code with generated assembly	C compiler option **--source**
Force definition of virtual function tables (C++)	C++ compiler option **--force-vtbl**
Suppress definition of virtual function tables (C++)	C++ compiler option **--suppress-vtbl**
Implicit inclusion of source files for finding templates	C++ compiler option **--implicit-include**
Minimal inlining of function calls (C++)	C++ compiler option **--no-inlining**
Instantiation mode of external template entities	C++ compiler option **--instantiate**
Comment in object file	C compiler option **--object-comment**
Additional options	C compiler options, C++ compiler options, Control program options

Assembler

Eclipse option	Description or option
Preprocessing	
Use TASKING preprocessor	Assembler option **--preprocessor-type**
Automatic inclusion of '.def' file	Assembler option **--include-file**

Eclipse option	Description or option
Defined symbols	Assembler option **--define**
Pre-include files	Assembler option **--include-file**
Include Paths	
Include paths	Assembler option **--include-directory**
Symbols	
Generate symbolic debug	Assembler option **--debug-info**
Case insensitive identifiers	Assembler option **--case-insensitive**
Emit local EQU symbols	Assembler option **--emit-locals=+equ**
Emit local non-EQU symbols	Assembler option **--emit-locals=+symbols**
Set default symbol scope to global	Assembler option **--symbol-scope**
Optimization	
Optimize generic instructions	Assembler option **--optimize=+generics**
Optimize instruction size	Assembler option **--optimize=+instr-size**
List File	
Generate list file	Control program option **--list-files**
List ...	Assembler option **--list-format**
List section summary	Assembler option **--section-info=+list**
Diagnostics	
Suppress warnings	Assembler option **--no-warnings=***num*
Suppress all warnings	Assembler option **--no-warnings**
Display section summary	Assembler option **--section-info=+console**
Maximum number of emitted errors	Assembler option **--error-limit**
Miscellaneous	
Allow Shift JIS Kanji in strings	Assembler option **--kanji**
Additional options	Assembler options

Linker

Eclipse option	Description or option
Output Format	
Generate Intel Hex format file	Linker option **--output=***file***:IHEX**
Generate S-records file	Linker option **--output=***file***:SREC**
Create file for each memory chip	Linker option **--chip-output**
Size of addresses (in bytes) for Intel Hex records	Linker option **--output=***file***:IHEX:***size*
Size of addresses (in bytes) for Motorola S records	Linker option **--output=***file***:SREC:***size*
Emit start address record	Linker option **--hex-format=s**

Eclipse option	Description or option
Hitex emulator (.htx) format (requires Hitex sptriced.exe)	Hitex sptriced.exe must be installed to generate this output format.
Libraries	
Link default libraries	Control program option **--no-default-libraries**
Rescan libraries to solve unresolved externals	Linker option **--no-rescan**
Libraries	The libraries are added as files on the command line.
Library search path	Linker option **--library-directory**
Data Objects	
Data objects	Linker option **--import-object**
Script File	
Defined symbols	Linker option **--define**
Linker script file	Linker option **--lsl-file**
Optimization	
Delete unreferenced sections	Linker option **--optimize=c**
Use a 'first-fit decreasing' algorithm	Linker option **--optimize=l**
Compress copy table	Linker option **--optimize=t**
Delete duplicate code	Linker option **--optimize=x**
Delete duplicate data	Linker option **--optimize=y**
Map File	
Generate map file (.map)	Control program option **--no-map-file**
Generate XML map file format (.mapxml) for map file viewer	Linker option **--map-file=***file***.mapxml:XML**
Include ...	Linker option **--map-file-format**
Diagnostics	
Suppress warnings	Linker option **--no-warnings=***num*
Suppress all warnings	Linker option **--no-warnings**
Maximum number of emitted errors	Linker option **--error-limit**
Miscellaneous	
Strip symbolic debug information	Linker option **--strip-debug**
Link case insensitive	Linker option **--case-insensitive**
Do not use standard copy table for initialization	Linker option **--user-provided-initialization-code**
Show link phases during processing	Linker option **--verbose**
Additional options	Linker options

10.1. Configuring the Command Line Environment

If you want to use the tools on the command line (either using a Windows command prompt or using Solaris), you can set *environment variables*.

You can set the following environment variables:

Environment variable	Description
ASTCINC	With this variable you specify one or more additional directories in which the assembler looks for include files. See Section 6.3, *How the Assembler Searches Include Files*.
CTCCACHE	With this variable you specify a cache directory in which the C compiler can store intermediate results. See C compiler option: --cache.
CTCINC	With this variable you specify one or more additional directories in which the C compiler looks for include files. See Section 4.4, *How the Compiler Searches Include Files*.
CPTCINC	With this variable you specify one or more additional directories in which the C++ compiler looks for include files. See Section 5.2, *How the C++ Compiler Searches Include Files*.
CCTCBIN	When this variable is set, the control program prepends the directory specified by this variable to the names of the tools invoked.
LIBTC1V1_3 / LIBTC1V1_3_1 / LIBTC1V1_6 / LIBTC1V1_6_X	With these variables you specify one or more additional directories in which the linker looks for libraries. See Section 7.3.1, *How the Linker Searches Libraries*.
PATH	With this variable you specify the directory in which the executables reside. This allows you to call the executables when you are not in the `bin` directory. Usually your system already uses the PATH variable for other purposes. To keep these settings, you need to add (rather than replace) the path. Use a semicolon (;) to separate path names.
TMPDIR	With this variable you specify the location where programs can create temporary files. Usually your system already uses this variable. In this case you do not need to change it.

See the documentation of your operating system on how to set environment variables.

10.2. C Compiler Options

This section lists all C compiler options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the compiler via the control program. Therefore, it uses the syntax of the control program to pass options and files to the C compiler. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » Miscellaneous**.

4. In the **Additional options** field, enter one or more command line options.

    *Because Eclipse uses the control program, you have to precede the option with **-Wc** to pass the option via the control program directly to the C compiler.*

Be aware that some command line options are not useful in Eclipse or just do not have any effect. For example, the option **-n** sends output to stdout instead of a file and has no effect in Eclipse.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+**-*longflag*. To switch a flag off, use an uppercase letter or a **-**-*longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
ctc -Oac test.c
ctc --optimize=+coalesce,+cse test.c
```

When you do not specify an option, a default value may become active.

C compiler option: --align

Menu entry

1. Select **C/C++ Compiler » Code Generation**.

2. Enter a value in the **Minimum alignment** field.

Command line syntax

--align=*value*

Default: **--align=0**

Description

By default the C compiler aligns objects to the minimum alignment required by the architecture. With this option you can increase this alignment for objects of four bytes or larger. The *value* must be a power of two and larger than or equal to 4.

Example

To align all objects of four bytes or larger on a 4-byte boundary, enter:

```
ctc --align=4 test.c
```

Instead of this option you can also specify the following pragma in your C source:

```
#pragma align 4
```

With `#pragma align restore` you can return to the previous alignment setting.

Related information

Pragma `align`

C compiler option: --cache

Menu entry

1. Select **C/C++ Compiler » Optimization » Compilation Speed**.

2. Enable the option **Cache generated code to improve the compilation speed**.

3. In the **Directory for cached files** field, enter the name for the location of the cache.

Command line syntax

--cache[**=***directory*]

Default on command line: . (current directory)

Default in Eclipse: .cache directory under project directory

Description

This option enables a cache for output files in the specified *directory*. When the source code after preprocessing and relevant compiler options and the compiler version are the same as in a previous invocation, the previous result is copied to the output file. The cache only works when there is a single C input file and a single output file (no **--mil-split**).

You can also enable the cache and specify the cache directory with the environment variable CTCCACHE. This option takes precedence over the environment variable.

The cache directory may be shared, for instance by placing it on a network drive. You can control the maximum size and/or age of the cache directory with the separate expiration tool **expiretc**.

The compiler creates a directory ctccache in the directory specified with the option **--cache** or the environment variable CTCCACHE. The directory is only created when it does not yet exist. The cache files are stored in this directory.

Example

To improve the compilation speed and put cached files in directory .cache, enter:

```
ctc --cache=.cache test.c
```

Related information

Section 11.6, *Compiler Cache*

Section 8.8, *Expire Cache Utility*

C compiler option: --cert

Menu entry

1. Select **C/C++ Compiler » CERT C Secure Coding**.

2. Make a selection from the **CERT C secure code checking** list.

3. If you selected **Custom**, expand the **Custom CERT C** entry and enable one or more individual recommendations/rules.

Command line syntax

```
--cert={all | name[-name],...}
```

Default format: all

Description

With this option you can enable one or more checks for CERT C Secure Coding Standard recommendations/rules. When you omit the argument, all checks are enabled. *name* is the name of a CERT recommendation/rule, consisting of three letters and two digits. Specify only the three-letter mnemonic to select a whole category. For the list of names you can use, see Chapter 19, *CERT C Secure Coding Standard*.

On the command line you can use **--diag=cert** to see a list of the available checks, or you can use a three-letter mnemonic to list only the checks in a particular category. For example, **--diag=pre** lists all supported preprocessor checks.

Example

To enable the check for CERT rule STR30-C, enter:

```
ctc --cert=str30 test.c
```

Related information

Chapter 19, *CERT C Secure Coding Standard*

C compiler option **--diag** (Explanation of diagnostic messages)

C compiler option: --check

Menu entry

-

Command line syntax

`--check`

Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application because the code will not actually be compiled.

The compiler reports any warnings and/or errors.

This option is available on the command line only.

Related information

Assembler option **--check** (Check syntax)

C compiler option: --code-core-association

Menu entry

1. Select **C/C++ Compiler » Allocation**.

2. Select a **Code core association**.

Command line syntax

--code-core-association=*value*

You can specify the following arguments:

clone	Multiple code instances, each executed by one core
private*n*	One code instance executed by core *n*. *n* can be 0 .. 7, depending on the available cores.
share	One code instance shared between cores.

Description

If a TriCore derivative has multiple cores you can use this option to specify the default core associations for functions.

This is option is available for multi-core TriCore architectures only.

Example

To specify that code instances can only be executed by core 0, enter:

```
ctc --core=tc1.6.x --code-core-association=private0 test.c
```

Related information

Section 1.4.2, *Code Core Association*

C compiler option **--data-core-association** (Data core association)

C compiler option: --code-section-alignment

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--code-section-alignment** to the **Additional options** field.

Command line syntax

`--code-section-alignment=`*value*

Default: 2

Description

By default the TriCore C compiler aligns code sections to the minimum alignment required by the architecture, which equals 2 bytes. With this option you can increase this alignment for code sections. The value must be a power of two and larger or equal 2. Absolute sections are not aligned.

Related information

C compiler option **--data-section-alignment** (Align data sections)

C compiler option: --compact-max-size

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. In the **Maximum size for code compaction** field, enter the maximum size of a match.

Command line syntax

```
--compact-max-size=value
```

Default: 200

Description

This option is related to the compiler optimization **--optimize=+compact** (Code compaction or reverse inlining). Code compaction is the opposite of inlining functions: large sequences of code that occur more than once, are transformed into a function. This reduces code size (possibly at the cost of execution speed).

However, in the process of finding sequences of matching instructions, compile time and compiler memory usage increase quadratically with the number of instructions considered for code compaction. With this option you tell the compiler to limit the number of matching instructions it considers for code compaction.

Example

To limit the maximum number of instructions in functions that the compiler generates during code compaction:

```
ctc --optimize=+compact --compact-max-size=100 test.c
```

Related information

C compiler option **--optimize=+compact** (Optimization: code compaction)

C compiler option **--max-call-depth** (Maximum call depth for code compaction)

C compiler option: --control-flow-info

Menu entry

1. Select **C/C++ Compiler » Debugging**.

2. Enable the option **Generate control flow information**.

Command line syntax

```
--control-flow-info
```

Description

With this option the compiler adds control flow information to the output file. The compiler generates a `.debug_control_flow` section which describes the basic blocks and their relations. This information can be used for code coverage analysis on optimized code.

Example

```
ctc --control-flow-info test.c
```

Related information

Section 8.6.2, *HLL Dump Output Format*

C compiler option **--debug-info** (Debug information)

C compiler option: --core

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor Selection** list, select a processor or select **User defined TriCore ...**.

Command line syntax

--core=*core*

You can specify the following *core* arguments:

tc1.3	TriCore 1.3 architecture
tc1.3.1	TriCore 1.3.1 architecture, TriCore 1.3.1 instructions may be generated
tc1.6	TriCore 1.6 architecture, TriCore 1.6 instructions may be generated
tc1.6.x	TriCore 1.6.x architecture, TriCore 1.6.x instructions may be generated
tc1.6.2	TriCore 1.6.2 architecture, TriCore 1.6.2 instructions may be generated

Default: **tc1.3**

Description

With this option you specify the core architecture for a target processor for which you create your application. If you use Eclipse or the control program, the TriCore toolset derives the core from the processor you selected. When the default core (**tc1.3**) is used, the macro __CORE_TC13__ is defined in the C source file.

With **--core=tc1.3.1**, the compiler can generate TriCore 1.3.1 instructions in the assembly file. The macro __CORE_TC131__ is defined in the C source file.

With **--core=tc1.6**, the compiler can generate TriCore 1.6 instructions in the assembly file. The macro __CORE_TC16__ is defined in the C source file.

With **--core=tc1.6.x**, the compiler can generate TriCore 1.6.x instructions in the assembly file. The macro __CORE_TC16X__ is defined in the C source file.

With **--core=tc1.6.2**, the compiler can generate TriCore 1.6.2 instructions in the assembly file. The macro __CORE_TC162__ is defined in the C source file.

Example

Specify a custom core:

```
ctc --core=tc1.3.1 test.c
```

Related information

Control program option **--cpu** (Select processor)

C compiler option **--mmu-present** (MMU present)

C compiler option **--fp-model=+soft** (Disable hardware floating-point instructions)

C compiler option: --data-core-association

Menu entry

1. Select **C/C++ Compiler » Allocation**.

2. Select a **Data core association**.

Command line syntax

--data-core-association=*value*

You can specify the following arguments:

clone	Multiple data instances, each accessed by one core
private*n*	One data instance accessed by core *n*. *n* can be 0 .. 7, depending on the available cores.
share	One data instance shared between cores.

Description

If a TriCore derivative has multiple cores you can use this option to specify the default core associations for data objects.

This is option is available for multi-core TriCore architectures only.

Example

To specify that data instances can only be accessed by core 1, enter:

```
ctc --core=tc1.6.x --data-core-association=private1 test.c
```

Related information

Section 1.4.1, *Data Core Association*

C compiler option **--code-core-association** (Code core association)

C compiler option: --data-section-alignment

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--data-section-alignment** to the **Additional options** field.

Command line syntax

`--data-section-alignment=`*value*

Default: 0

Description

By default the data sections data objects are allocated in do not require an alignment, because the data objects are individually aligned to the minimum alignment required by the architecture. With this option you can increase the data section alignment. The value must be a power of two.

This option is only required to support backward compatible data section generation. Absolute sections are not aligned.

Related information

C compiler option **--code-section-alignment** (Align code sections)

C compiler option: --debug-info (-g)

Menu entry

1. Select **C/C++ Compiler » Debugging**.

2. To **generate symbolic debug information**, select **Default**, **Small set** or **Full**.
   To disable the generation of debug information, select **None**.

Command line syntax

--debug-info[**=***suboption*]

-g[*suboption*]

You can set the following suboptions:

small	**1** | **c**	Emit small set of debug information.
default	**2** | **d**	Emit default symbolic debug information.
all	**3** | **a**	Emit full symbolic debug information.

Default: **--debug-info** (same as **--debug-info=default**)

Description

With this option you tell the compiler to add directives to the output file for including symbolic information. This facilitates high level debugging but increases the size of the resulting assembler file (and thus the size of the object file). For the final application, compile your C files without debug information.

The DWARF debug format allows for a flexible approach as to how much symbolic information is included, as long as the structure is valid. Adding all possible DWARF data for a program is not practical. The amount of DWARF information per compilation unit can be huge. And for large projects, with many object modules the link time can grow unacceptably long. That is why the compiler has several debug information levels. In general terms one can say, the higher the level the more DWARF information is produced.

The DWARF data in an object module is not only used for debugging. The toolset can also do "type checking" of the whole application. In that case the linker will use the DWARF information of all object modules to determine if every use of a symbol is done with the same type. In other words, if the application is built with type checking enabled then the compiler will add DWARF information too.

Small set of debug information

With this suboption only DWARF call frame information and type information are generated. This enables you to inspect parameters of nested functions. The type information improves debugging. You can perform a stack trace, but stepping is not possible because debug information on function bodies is not generated. You can use this suboption, for example, to compact libraries.

Default debug information

This provides all debug information you need to debug your application. It meets the debugging requirements in most cases without resulting in oversized assembler/object files.

Full debug information

With this suboption extra debug information is generated about unused typedefs and DWARF "lookup table sections". Under normal circumstances this extra debug information is not needed to debug the program. Information about unused typedefs concerns all typedefs, even the ones that are not used for any variable in the program. (Possibly, these unused typedefs are listed in the standard include files.) With this suboption, the resulting assembler/object file will increase significantly.

In the following table you see in more detail what DWARF information is included for the debug option levels.

Feature	-g1	-g2	-g3	type check	Remarks
basic info	+	+	+	+	info such as symbol name and type
call frame	+	+	+	+	this is information for a debugger to compute a stack trace when a program has stopped at a breakpoint
symbol lifetime		+	+		this is information about where symbols live (e.g. on stack at offset so and so, when the program counter is in this range)
line number info		+	+	+	file name, line number, column number
"lookup tables"			+		DWARF sections ... this is an optimization for the DWARF data, it is not essential
unused typedefs			+		in the C/C++ code of the program there can be (many) typedefs that are not used for any variable. Sometimes this can cause enormous expansion of the DWARF data and thus it is only included in **-g3**.

Related information

-

C compiler option: --default-a0-size (-Z)

Menu entry

1. Select **C/C++ Compiler » Allocation**

2. In the **Threshold for putting data in __a0** field, enter a value in bytes.

Command line syntax

--default-a0-size[**=***threshold*]
-Z[*threshold*]

Default: **--default-a0-size=0**

Description

With this option you can specify a threshold value for __a0 allocation. If you do not specify a memory qualifier such as __near or __far in the declaration of an object, the compiler chooses where to place the object based on the size of the object.

First, the size of the object is checked against the near size threshold, according to the description of the **--default-near-size** (**-N**) option. If the size of the object is larger than the near size threshold, but lower or equal to the a0 size threshold, the object is allocated in __a0 memory. Larger objects, arrays and strings will be allocated __far.

By default the a0 size threshold is zero, which means that the compiler will never use __a0 memory unless you specify the **--default-a0-size** (**-Z**) option. If you omit a threshold value, all objects not allocated __near, including arrays and string constants, will be allocated in __a0 memory.

Allocation in __a0 memory means that the object is addressed indirectly, using A0 as the base pointer. The total amount of memory that can be addressed this way is 64 kB.

Instead of this option you can also use #pragma default_a0_size in the C source.

Example

To put all data objects with a size of 12 bytes or smaller, but larger than the default near threshold of 8, in __a0 sections:

```
ctc --default-a0-size=12 test.c
```

Related information

C compiler option **--default-a1-size (-Y)** (maximum size in bytes for data elements that are by default located in __a1 sections)

C compiler option **--default-near-size (-N)** (maximum size in bytes for data elements that are by default located in __near sections)

Section 1.2.1, *Memory Qualifiers*

C compiler option: --default-a1-size (-Y)

Menu entry

1. Select **C/C++ Compiler » Allocation**

2. In the **Threshold for putting data in __a1** field, enter a value in bytes.

Command line syntax

--default-a1-size[**=***threshold*]
-Y[*threshold*]

Default: **--default-a1-size=0**

Description

With this option you can specify a threshold value for __a1 allocation. If you do not specify a memory qualifier such as __near or __far in the declaration of an object, the compiler chooses where to place the object based on the size of the object.

First, the size of the object is checked against the near size threshold, according to the description of the **--default-near-size** (**-N**) option. If the size of the object is larger than the near size threshold, but lower or equal to the a1 size threshold, the object is allocated in __a1 memory. Larger objects, arrays and strings will be allocated __far.

By default the a1 size threshold is zero, which means that the compiler will never use __a1 memory unless you specify the **--default-a1-size** (**-Y**) option. If you omit a threshold value, all objects not allocated __near, including arrays and string constants, will be allocated in __a1 memory.

Allocation in __a1 memory means that the object is addressed indirectly, using A1 as the base pointer. The total amount of memory that can be addressed this way is 64 kB.

Instead of this option you can also use #pragma default_a1_size in the C source.

Example

To put all data objects with a size of 12 bytes or smaller, but larger than the default near threshold of 8, in __a1 sections:

ctc --default-a1-size=12 test.c

Related information

C compiler option **--default-a0-size (-Z)** (maximum size in bytes for data elements that are by default located in __a0 sections)

C compiler option **--default-near-size (-N)** (maximum size in bytes for data elements that are by default located in __near sections)

Section 1.2.1, *Memory Qualifiers*

C compiler option: --default-near-size (-N)

Menu entry

1. Select **C/C++ Compiler » Allocation**

2. In the **Threshold for putting data in __near** field, enter a value in bytes.

Command line syntax

--default-near-size[**=***threshold*]
-N[*threshold*]

Default: **--default-near-size=8**

Description

With this option you can specify a threshold value for __near allocation. If you do not specify __near or __far in the declaration of an object, the compiler chooses where to place the object. The compiler allocates objects smaller or equal to the threshold in __near sections. Larger objects are allocated in __a0, __a1 or __far sections.

If you omit a threshold value, all objects will be allocated __near, including arrays and string constants.

Instead of this option you can also use #pragma default_near_size in the C source.

Example

To put all data objects with a size of 12 bytes or smaller in __near sections:

```
ctc --default-near-size=12 test.c
```

Related information

C compiler option **--default-a0-size (-Z)** (maximum size in bytes for data elements that are by default located in __a0 sections)

C compiler option **--default-a1-size (-Y)** (maximum size in bytes for data elements that are by default located in __a1 sections)

Section 1.2.1, *Memory Qualifiers*

C compiler option: --define (-D)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

    The Defined symbols box shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, demo=1)

> Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

--define=*macro_name*[**=***macro_definition*]

-D*macro_name*[**=***macro_definition*]

Description

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, you can use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the compiler with the option **--option-file** (**-f**) *file*.

Defining macros with this option (instead of in the C source) is, for example, useful to compile conditional C source as shown in the example below.

> Make sure you do not use a reserved keyword as a macro name, as this can lead to unexpected results.

Example

Consider the following C program with conditional code to compile a demo program and a real program:

```
void main( void )
{
#if DEMO
    demo_func();   /* compile for the demo program */
#else
    real_func();   /* compile for the real program */
#endif
}
```

You can now use a macro definition to set the DEMO flag:

```
ctc --define=DEMO test.c
ctc --define=DEMO=1 test.c
```

Note that both invocations have the same effect.

The next example shows how to define a macro with arguments. Note that the macro name and definition are placed between double quotes because otherwise the spaces would indicate a new option.

```
ctc --define="MAX(A,B)=((A) > (B) ? (A) : (B))" test.c
```

Related information

C compiler option **--undefine** (Remove preprocessor macro)

C compiler option **--option-file** (Specify an option file)

C compiler option: --dep-file

Menu entry

Eclipse uses this option in the background to create a file with extension `.d` (one for every input file).

Command line syntax

`--dep-file`[**`=`***file*]

Description

With this option you tell the compiler to generate dependency lines that can be used in a Makefile. In contrast to the option **--preprocess=+make**, the dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension `.d` (one for every input file). When you specify a filename, all dependencies will be combined in the specified file.

Example

```
ctc --dep-file=test.dep test.c
```

The compiler compiles the file `test.c`, which results in the output file `test.src`, and generates dependency lines in the file `test.dep`.

Related information

C compiler option **--preprocess=+make** (Generate dependencies for make)

C compiler option: --diag

Menu entry

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

   The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

   A popup menu appears.

3. Select **Detailed Diagnostics Info**.

   A dialog box appears with additional information.

Command line syntax

--diag=[*format*:]{**all** | *msg*[*-msg*],...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. The compiler does not compile any files. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given (except for the CERT checks). If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas, or you can specify a range.

With **--diag=cert** you can see a list of the available CERT checks, or you can use a three-letter mnemonic to list only the checks in a particular category. For example, **--diag=pre** lists all supported preprocessor checks.

Example

To display an explanation of message number 282, enter:

```
ctc --diag=282
```

This results in the following message and explanation:

```
E282: unterminated comment
```

```
Make sure that every comment starting with /* has a matching */.
Nested comments are not possible.
```

To write an explanation of all errors and warnings in HTML format to file `cerrors.html`, use redirection and enter:

```
ctc --diag=html:all > cerrors.html
```

Related information

Section 4.8, *C Compiler Error Messages*

C compiler option **--cert** (Enable individual CERT checks)

C compiler option: --eabi

Command line syntax

`--eabi=``flags`

You can set the following flags:

+/-float	**f/F**	allow treat 'double' as 'float'
+/-half-word-align	**h/H**	allow half-word alignment
+/-no-clear	**n/N**	allow the use of option **--no-clear**
+/-structure-return	**s/S**	allow structure-return optimization

Default: **`--eabi=fhns`**

Description

With this option you control the level of EABI compliancy.

With **--eabi=+float**, the compiler is allowed to treat variables and constants of type double as float when option **--fp-model=+float** is used. Because the float type takes less space, execution speed increases and code size decreases, both at the cost of less precision. The TriCore FPU does support single precision float operations, but does not support double precision float operations. The EABI prescribes that the double precision float type is 8 bytes and conform IEEE-754 floating-point numbers. (EABI 2.1.2 Fundamental Data Types)

With **--eabi=+half-word-align**, alignment of 32-bit integer variables on half-word boundaries is allowed. The natural alignment of 32-bit integer variables is a word boundary, but generally the TriCore architecture supports access to 32-bit integer variables on half-word boundaries. (EABI 2.1.5.1 Discrete Word and Double Word Variables)

With **--eabi=+no-clear**, the use of option **--no-clear** and #pragma noclear is allowed.

With **--eabi=+structure-return**, if a function result is the right-hand side of a structure assignment, the function result can directly be copied to the global object at the left-hand side. This is an optimization that saves a structure copy from stack to global memory in the caller function. For EABI compliance it is only allowed that the called function returns the structure in a temporary object allocated on the stack by the caller before it is copied to the global object by the caller. (EABI 2.2.5.3 Structure Return Values) Only structures or unions larger than 64-bit are involved, because when smaller they are returned in a register.

Related information

C compiler option **--eabi-compliant** (code needs to be completely EABI compliant)

C compiler option **--fp-model** (floating-point model)

C compiler option **--no-clear** (do not clear non-initialized global/static variables)

C compiler option: --eabi-compliant

Command line syntax

`--eabi-compliant`

Description

Use this option when the generated code needs to be completely EABI compliant.

This option is an alias for **--eabi=FHNS**.

Related information

C compiler option **--eabi** (control level of EABI compliancy)

C compiler option: --error-file

Menu entry

-

Command line syntax

--error-file[**=***file*]

Description

With this option the compiler redirects error messages to a file. If you do not specify a filename, the error file will be named after the output file with extension `.err`.

Example

To write errors to `errors.err` instead of `stderr`, enter:

```
ctc --error-file=errors.err test.c
```

Related information

-

C compiler option: --fp-model

Menu entry

1. Select **C/C++ Compiler » Floating-Point**.

2. Make a selection from the **Floating-point model** list.

3. If you selected **Custom**, enable one or more individual options.

Command line syntax

`--fp-model=``flags`

You can set the following flags:

+/-contract	**c/C**	allow expression contraction
+/-float	**f/F**	treat 'double' as 'float'
+/-fastlib	**l/L**	allow less precise library functions
+/-nonan	**n/N**	allow optimizations to ignore NaN/Inf
+/-rewrite	**r/R**	allow expression rewriting
+/-soft	**s/S**	use software floating-point library
+/-trap	**t/T**	support trapping on exceptions
+/-negzero	**z/Z**	ignore sign of -0.0
	0	alias for **--fp-model=CFLNRStZ** (strict)
	1	alias for **--fp-model=cFLNRSTZ** (precise)
	2	alias for **--fp-model=cFlnrSTz** (fast double)
	3	alias for **--fp-model=cflnrSTz** (fast single)

Default: **`--fp-model=cFlnrSTz`**

Description

With this option you select the floating-point execution model.

With **--fp-model=+contract** you allow the compiler to contract multiple float operations into a single operation, with different rounding results. A possible example is fused multiply-add. With **--fp-model=-contract**, the fused multiply-and-accumulate (FMA) operations are not generated. FMA operations are not supported by the IEEE-754 standard. The result of FMA operations is only rounded once at the end of the FMA.

With **--fp-model=+float** you tell the compiler to treat variables and constants of type `double` as `float`. Because the `float` type takes less space, execution speed increases and code size decreases, both at the cost of less precision. Make sure you specify the corresponding libraries to the linker.

With **--fp-model=+fastlib** you allow the compiler to select faster but less accurate library functions for certain floating-point operations. With **--fp-model=-fastlib** more precise library functions are used and the compiler defines the macro `__PRECISE_LIB_FP__`, which is used in `math.h`.

With **--fp-model=+nonan** you allow the compiler to ignore NaN or Inf input values. An example is to replace multiply by zero with zero.

With **--fp-model=+rewrite** you allow the compiler to rewrite expressions by reassociating. This might result in rounding differences and possibly different exceptions. An example is to rewrite (a*c)+(b*c) as (a+b)*c.

With **--fp-model=+soft** no hardware floating-point instructions are generated, only calls to the software floating-point library.

By default, the compiler assumes an FPU is present and the macro `__FPU__` is defined in the C source file. With this option the compiler does not generate single precision floating-point instructions in the assembly file. When you select this option, the macro `__FPU__` is not defined in the C source file and the compiler generates the assembler control `$NO_FPU` in the assembly source.

With **--fp-model=+trap** operations trap on floating-point exceptions. The floating-point instructions, as implemented in the FPU, need to be handled in a special way if floating-point trapping behavior is expected from the generated code. Make sure you specify the corresponding trapping floating-point library to the linker.

With **--fp-model=+negzero** you allow the compiler to ignore the sign of -0.0 values. An example is to replace (a-a) by zero.

Related information

Pragmas `STDC FP_CONTRACT`, `fp_negzero`, `fp_nonan` and `fp_rewrite` in Section 1.8, *Pragmas to Control the Compiler*.

C compiler option **--eabi=+float** (allow treat 'double' as 'float')

C compiler option: --global-type-checking

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

2. Enable the option **Perform global type checking on C code**.

Command line syntax

```
--global-type-checking
```

Description

The C compiler already performs type checking within each module. Use this option when you want the linker to perform type checking between modules.

Related information

-

C compiler option: --help (-?)

Menu entry

-

Command line syntax

--help[**=***item*]

-?

You can specify the following arguments:

intrinsics	**i**	Show the list of intrinsic functions
options	**o**	Show extended option descriptions
pragmas	**p**	Show the list of supported pragmas
typedefs	**t**	Show the list of predefined typedefs

Description

Displays an overview of all command line options. With an argument you can specify which extended information is shown.

Example

The following invocations all display a list of the available command line options:

```
ctc -?
ctc --help
ctc
```

The following invocation displays a list of the available pragmas:

```
ctc --help=pragmas
```

Related information

-

C compiler option: --immediate-in-code

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--immediate-in-code** to the **Additional options** field.

Command line syntax

```
--immediate-in-code
```

Description

By default the TriCore C compiler creates a data object to represent an immediate value of 32 or 64 bits, then loading this constant value directly into a register. With this option you can tell the compiler to code the immediate values directly into the instructions, thus using less data, but more code.

Actually when option **--default-near-size** < 4, 32-bit immediates will be coded into instructions anyhow, when it is >= 4 they will be located in neardata. When **--default-near-size** < 8, 64-bit immediates will be located in fardata, when it is >= 8 they will be located in neardata as well.

Instead of this option you can also specify the following pragma in your C source:

```
#pragma immediate_in_code
```

Related information

C compiler option **--default-near-size** (Maximum size for objects to be allocated by default in `__near` sections)

Pragma `immediate_in_code`

C compiler option: --include-directory (-I)

Menu entry

1.   Select **C/C++ Compiler » Include Paths**.

    The Include paths box shows the directories that are added to the search path for include files.

2.   To define a new directory for the search path, click on the **Add** button in the **Include paths** box.

3.   Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

--include-directory=*path*,...

-I*path*,...

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory,

The order in which the compiler searches for include files is:

1. The pathname in the C source file and the directory of the C source (only for #include files that are enclosed in "")

2. The path that is specified with this option.

3. The path that is specified in the environment variable CTCINC when the product was installed.

4. The default directory $(PRODDIR)\include (unless you specified option **--no-stdinc**).

Example

Suppose that the C source file test.c contains the following lines:

```
#include <stdio.h>
#include "myinc.h"
```

You can call the compiler as follows:

```
ctc --include-directory=myinclude test.c
```

First the compiler looks for the file stdio.h in the directory myinclude relative to the current directory. If it was not found, the compiler searches in the environment variable and then in the default include directory.

The compiler now looks for the file `myinc.h` in the directory where `test.c` is located. If the file is not there the compiler searches in the directory `myinclude`. If it was still not found, the compiler searches in the environment variable and then in the default include directory.

Related information

C compiler option **--include-file** (Include file at the start of a compilation)

C compiler option **--no-stdinc** (Skip standard include files directory)

C compiler option: --include-file (-H)

Menu entry

1.   Select **C/C++ Compiler » Preprocessing**.

     The Pre-include files box shows the files that are currently included before the compilation starts.

2.   To define a new file, click on the **Add** button in the **Pre-include files** box.

3.   Type the full path and file name or select a file.

Use the **Edit** and **Delete** button to change a file name or to remove a file from the list.

Command line syntax

```
--include-file=file,...
```

```
-Hfile,...
```

Description

With this option you include one or more extra files at the beginning of each C source file, before other includes. This is the same as specifying #include "file" at the beginning of *each* of your C sources.

Example

```
ctc --include-file=stdio.h test1.c test2.c
```

The file stdio.h is included at the beginning of both test1.c and test2.c.

Related information

C compiler option **--include-directory** (Add directory to include file search path)

C compiler option: --indirect

Menu entry

1. Select **C/C++ Compiler » Code Generation**.

2. Enable the option **Call functions indirectly**.

Command line syntax

--indirect

Description

With this option you tell the compiler to generate code for indirect function calling.

Instead of this option you can also specify the following pragma in your C source:

```
#pragma indirect
```

Example

With the following command the compiler generates far calls for all functions:

```
ctc --indirect test.c
```

Related information

C compiler option **--indirect-runtime** (Call run-time functions indirectly)

Section 1.11.1, "Function calling modes: __indirect"

C compiler option: --indirect-runtime

Menu entry

1. Select **C/C++ Compiler » Code Generation**.

2. Enable the option **Call run-time functions indirectly**.

Command line syntax

`--indirect-runtime`

Description

With this option you tell the compiler to generate code for indirect calls to run-time functions. Use this option if you locate the entire run-time library in far memory.

Instead of this option you can also specify the following pragma in your C source:

```
#pragma indirect_runtime
```

Example

With the following command the compiler generates far calls for all run-time functions:

```
ctc --indirect-runtime test.c
```

Related information

C compiler option **--indirect** (Call functions indirectly)

Section 1.11.1, "Function calling modes: __indirect"

C compiler option: --inline

Menu entry

1.   Select **C/C++ Compiler » Optimization**.

2.   Enable the option **Always inline function calls**.

Command line syntax

```
--inline
```

Description

With this option you instruct the compiler to inline calls to functions without the `__noinline` function qualifier whenever possible. This option has the same effect as a `#pragma inline` at the start of the source file.

This option can be useful to increase the possibilities for code compaction (C compiler option **--optimize=+compact**).

Example

To always inline function calls:

```
ctc --optimize=+compact --inline test.c
```

Related information

C compiler option **--optimize=+compact** (Optimization: code compaction)

Section 1.11.3, *Inlining Functions: inline*

C compiler option: --inline-max-incr / --inline-max-size

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. In the **Maximum size increment when inlining** field, enter a value (default -1).

3. In the **Maximum size for functions to always inline** field, enter a value (default -1).

Command line syntax

```
--inline-max-incr=percentage  (default: -1)
--inline-max-size=threshold   (default: -1)
```

Description

With these options you can control the automatic function inlining optimization process of the compiler. These options only have effect when you have enabled the inlining optimization (option **--optimize=+inline** or **Optimize most**).

> Regardless of the optimization process, the compiler always inlines all functions that have the function qualifier `inline`.

With the option **--inline-max-size** you can specify the maximum size of functions that the compiler inlines as part of the optimization process. The compiler always inlines all functions that are smaller than the specified *threshold*. The threshold is measured in compiler internal units and the compiler uses this measure to decide which functions are small enough to inline. The default threshold is -1, which means that the threshold depends on the option **--tradeoff**.

After the compiler has inlined all functions that have the function qualifier `inline` and all functions that are smaller than the specified threshold, the compiler looks whether it can inline more functions without increasing the code size too much. With the option **--inline-max-incr** you can specify how much the code size is allowed to increase. The default value is -1, which means that the value depends on the option **--tradeoff**.

Example

```
ctc --optimize=+inline --inline-max-incr=40 --inline-max-size=15 test.c
```

The compiler first inlines all functions with the function qualifier `inline` and all functions that are smaller than the specified threshold of 15. If the code size has still not increased with 40%, the compiler decides which other functions it can inline.

Related information

C compiler option **--optimize=+inline** (Optimization: automatic function inlining)
Section 1.11.3, *Inlining Functions: inline*
Section 4.6.3, *Optimize for Code Size or Execution Speed*

C compiler option: --integer-enumeration

Menu entry

1.   Select **C/C++ Compiler » Language**.

2.   Enable the option **Always use 32-bit integers for enumeration**.

Command line syntax

`--integer-enumeration`

Description

Normally the compiler treats enumerated types as the smallest data type possible (`char` or `short` instead of `int`). This reduces code size. With this option the compiler always treats enum-types as `int` as defined in the ISO C99 standard.

Related information

Section 1.1, *Data Types*

C compiler option: --iso (-c)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. From the **Comply to C standard** list, select **ISO C99** or **ISO C90**.

Command line syntax

`--iso={90|99}`

`-c{90|99}`

Default: `--iso=99`

Description

With this option you select the ISO C standard. C90 is also referred to as the "ANSI C standard". C99 refers to the newer ISO/IEC 9899:1999 (E) standard. C99 is the default.

Example

To select the ISO C90 standard on the command line:

```
ctc --iso=90 test.c
```

Related information

C compiler option **--language** (Language extensions)

C compiler option: --keep-output-files (-k)

Menu entry

Eclipse *always* removes the `.src` file when errors occur during compilation.

Command line syntax

```
--keep-output-files
```

```
-k
```

Description

If an error occurs during compilation, the resulting `.src` file may be incomplete or incorrect. With this option you keep the generated output file (`.src`) when an error occurs.

By default the compiler removes the generated output file (`.src`) when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to inspect the generated assembly source. Even if it is incomplete or incorrect.

Example

```
ctc --keep-output-files test.c
```

When an error occurs during compilation, the generated output file `test.src` will *not* be removed.

Related information

C compiler option **--warnings-as-errors** (Treat warnings as errors)

C compiler option: --language (-A)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable or disable one or more of the following options:

   - Allow GNU C extensions

   - Allow // comments in ISO C90 mode

   - Check assignment of string literal to non-const string pointer

   - Allow optimization across volatile access

   - Allow Shift JIS Kanji in strings

Command line syntax

--language=[*flags*]

-A[*flags*]

You can set the following flags:

+/-gcc	**g/G**	enable a number of gcc extensions
+/-kanji	**k/K**	support for Shift JIS Kanji in strings
+/-comments	**p/P**	// comments in ISO C90 mode
+/-volatile	**v/V**	don't optimize across volatile access
+/-strings	**x/X**	relaxed const check for string literals

Default: **-AGKpVx**

Default (without flags): **-AGKPVX**

Description

With this option you control the language extensions the compiler can accept. By default the TriCore compiler allows all language extensions, except for **gcc** extensions.

The option **--language** (**-A**) without flags disables all language extensions.

GNU C extensions

The **--language=+gcc** (**-Ag**) option enables the following gcc language extensions:

- The identifier __FUNCTION__ expands to the current function name.

- Alternative syntax for variadic macros.

- Alternative syntax for designated initializers.

- Allow zero sized arrays.

- Allow empty struct/union.

- Allow unnamed struct/union fields.

- Allow empty initializer list.

- Allow initialization of static objects by compound literals.

- The middle operand of a ? : operator may be omitted.

- Allow a compound statement inside braces as expression.

- Allow arithmetic on void pointers and function pointers.

- Allow a range of values after a single case label.

- Additional preprocessor directive #warning.

- Allow comma operator, conditional operator and cast as lvalue.

- An inline function without "static" or "extern" will be global.

- An "extern inline" function will not be compiled on its own.

- An __attribute__ directly following a struct/union definition relates to that tag instead of to the objects in the declaration.

For a more complete description of these extensions, you can refer to the UNIX gcc info pages (**info gcc**).

Shift JIS Kanji support

With **--language=+kanji** (**-Ak**) you tell the compiler to support Shift JIS encoded Kanji multi-byte characters in strings, (wide) character constants and // comments. Without this option, encodings with 0x5c as the second byte conflict with the use of the backslash as an escape character. Shift JIS in /*...*/ comments is supported regardless of this option. Note that Shift JIS also includes Katakana and Hiragana.

Comments in ISO C90 mode

With **--language=+comments** (**-Ap**) you tell the compiler to allow C++ style comments (//) in ISO C90 mode (option **--iso=90**). In ISO C99 mode this style of comments is always accepted.

Check assignment of string literal to non-const string pointer

With **--language=+strings** (**-Ax**) you disable warnings about discarded const qualifiers when a string literal is assigned to a non-const pointer.

```
char *p;
void main( void ) { p = "hello"; }
```

Example

```
ctc --language=-comments,+strings --iso=90 test.c
ctc -APx -c90 test.c
```

The compiler compiles in ISO C90 mode, accepts assignments of a constant string to a non-constant string pointer and does not allow C++ style comments.

Optimization across volatile access

With the **--language=+volatile** (**-Av**) option, the compiler will block optimizations when reading or writing a volatile object, by executing all memory and (SFR) register accesses before the access of the volatile object. The volatile access acts as a memory barrier. With this option you can prevent for example that code below the volatile object is optimized away to somewhere above the volatile object.

Example:

```
extern unsigned int variable;
extern volatile unsigned int access;

void TestFunc( unsigned int flag )
{
    access = 0;
    variable |= flag;
    if( variable == 3 )
    {
        variable = 0;
    }
    variable |= 0x8000;
    access = 1;
}
```

Result with **--language=-volatile** (default):

```
_TestFunc   .type   func
    mov     d15,#0
    ld.w    d0,variable    ; <== Moved across volatile access
    or      d0,d4
    st.w    access,d15      ; <== Volatile access
    eq      d15,d0,#3
    cmov    d0,d15,#0
    insert  d15,d0,#1,#15,#1
    st.w    variable,d15
    mov     d0,#1
    st.w    access,d0       ; <== Volatile access
    ret
```

Result with **--language=+volatile**:

```
_TestFunc   .type   func
    mov     d15,#0
    st.w    access,d15      ; <== Volatile access
```

```
ld.w     d0,variable     ; <== Not moved
or       d0,d4
eq       d15,d0,#3
cmov     d0,d15,#0
insert   d15,d0,#1,#15,#1
st.w     variable,d15
mov      d15,#1
st.w     access,d15      ; <== Volatile access
ret
```

Note that the volatile behavior of the compiler with option **--language=-volatile** or **--language=+volatile** is ISO C compliant in both cases.

Related information

C compiler option **--iso** (ISO C standard)

Section 1.5, *Shift JIS Kanji Support*

C compiler option: --loop

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--loop** to the **Additional options** field.

Command line syntax

--loop[**=***flags*]

You can set the following flags:

+/-cache	**c/C**	Align loops for cached memory or SPRAM
+/-fusion	**f/F**	Loop fusion optimization
+/-instructions	**i/I**	Use loop body instruction threshold
+/-size	**s/S**	Use loop body size threshold
+/-time	**t/T**	Use loop body cycle threshold
+/-value	**v/V**	Use alignment value or memory default

Default: **--loop=cfistV**

Description

With this option you control loop optimizations of the compiler.

Align loops for cached memory or SPRAM

With **--loop=+cache** you tell the compiler to align loops that are located in cached memory or SPRAM. With **--loop=-cache** loops are aligned for non-cached memory in segment 0xa. Only for TriCore 1.3 and 1.3.1.

Loop fusion optimization

With **--loop=+fusion** you tell the compiler to fuse loops when the loop transformation optimization is enabled (**--optimize=+loop**). This means that the compiler optimizes loops into one loop when possible.

Example:

```
int var1;
int var2;
int Arr1[ 8 ];
int Arr2[ 8 ];
void foo( void )
{
    int i;
    var1 = 0;
    for ( i = 0; i < 8; i++ )
```

```
    {
        var1 += Arr1[ i ];
    }
    var2 = 0;
    for ( i = 0; i < 8; i++ )
    {
        var2 += Arr2[ i ];
    }
}
```

With loop fusion enabled (the default) this source is compiled to:

```
foo:  .type   func
      mov     d15,#0
      movh.a  a15,#@his(Arr1)
      mov     d0,d15
      movh.a  a2,#@his(Arr2)
      lea     a15,[a15]@los(Arr1)
      lea     a2,[a2]@los(Arr2)
      mov.a   a4,#7
.L2:
      ld.w    d1,[a15+]
      add     d15,d1
      ld.w    d1,[a2+]
      add     d0,d1
      loop    a4,.L2       ; one loop instead of two
      st.w    var2,d0
      st.w    var1,d15
      ret
```

With loop fusion disabled (**--loop=-fusion**) this source is compiled to:

```
foo:  .type   func
      mov     d15,#0
      movh.a  a15,#@his(Arr1)
      lea     a15,[a15]@los(Arr1)
      mov.a   a2,#7
.L2:
      ld.w    d0,[a15+]
      add     d15,d0
      loop    a2,.L2       ; loop one
      mov     d0,#0
      movh.a  a15,#@his(Arr2)
      lea     a15,[a15]@los(Arr2)
      mov.a   a2,#7
.L3:
      ld.w    d1,[a15+]
      add     d0,d1
      loop    a2,.L3       ; loop two
      st.w    var2,d0
```

```
st.w    var1,d15
ret
```

Loop body instruction threshold

With **--loop=+instructions** you tell the compiler to align loops that have a loop body with a number of instructions that is smaller than the number of instruction specified with option **--loop-instruction-threshold**.

Loop body size threshold

With **--loop=+size** you tell the compiler to align loops that have a loop body size in bytes that is smaller than the number of bytes specified with option **--loop-size-threshold**.

Loop body cycle threshold

With **--loop=+time** you tell the compiler to align loops that have an estimated cycle count of its body that is smaller than the number of cycles specified with option **--loop-cycle-threshold**.

Fixed loop alignment

With **--loop=+value** you tell the compiler to use a fixed loop alignment value for the alignment of loops. The alignment value is specified with option **--loop-alignment=***value*.

Related information

C compiler option **--optimize=+align-loop** (Optimization: align loop bodies)

C compiler option **--optimize=+loop** (Optimization: loop transformations)

C compiler option **--loop-alignment** (Alignment value of loop bodies)

C compiler option **--loop-cycle-threshold** (Loop body cycle threshold)

C compiler option **--loop-instruction-threshold** (Loop body instruction threshold)

C compiler option **--loop-size-threshold** (Loop body byte size threshold)

C compiler option: --loop-alignment

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--loop-alignment** to the **Additional options** field.

Command line syntax

```
--loop-alignment=alignment
```

Default: 32

Description

Specify the alignment loop bodies will get when the loop-alignment optimization is used. When a loop body is properly aligned, the number of fetches required to retrieve the loop body, can be reduced significantly. By default the compiler uses the best alignment, which is overruled when **--loop=+value** is enabled. Then the loops will get the alignment specified with this option. This alignment can be either 4 bytes, 8 bytes, 16 bytes or 32 bytes. The default alignment value is 32 bytes, because this is equal to a 256-bit fetch line. Loops are only aligned if the align-loop optimization is enabled and the tradeoff is set to speed (<=2)

Instead of this option you can also specify the following pragma in your C source:

```
#pragma loop_alignment value
```

Related information

C compiler option **--loop=+value** (Use alignment value or memory default)

C compiler option **--optimize=+align-loop** (Optimization: align loop bodies)

Pragma `loop_alignment`

C compiler option: --loop-cycle-threshold

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--loop-cycle-threshold** to the **Additional options** field.

Command line syntax

`--loop-cycle-threshold=`*value*

Default: 25

Description

When **--optimize=+align-loop** and **--loop=+time** are enabled and a loop body has an estimated cycle count that is smaller than this threshold it is optimized for alignment.

Related information

C compiler option **--loop=+time** (Use loop body cycle threshold)

C compiler option **--optimize=+align-loop** (Optimization: align loop bodies)

C compiler option: --loop-instruction-threshold

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--loop-instruction-threshold** to the **Additional options** field.

Command line syntax

`--loop-instruction-threshold=`*value*

Default: 25

Description

When **--optimize=+align-loop** and **--loop=+instructions** are enabled and a loop body has an instruction that is smaller than this threshold it is optimized for alignment.

Related information

C compiler option **--loop=+instruction** (Use loop body instruction threshold)

C compiler option **--optimize=+align-loop** (Optimization: align loop bodies)

C compiler option: --loop-size-threshold

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--loop-size-threshold** to the **Additional options** field.

Command line syntax

```
--loop-size-threshold=value
```

Default: 75

Description

When **--optimize=+align-loop** and **--loop=+size** are enabled and a loop body has a size in bytes that is smaller than this threshold it is optimized for alignment.

Related information

C compiler option **--loop=+size** (Use loop body byte size threshold)

C compiler option **--optimize=+align-loop** (Optimization: align loop bodies)

C compiler option: --make-target

Menu entry

-

Command line syntax

`--make-target=`*name*

Description

With this option you can overrule the default target name in the make dependencies generated by the options **--preprocess=+make** (**-Em**) and **--dep-file**. The default target name is the basename of the input file, with extension `.o`.

Example

```
ctc --preprocess=+make --make-target=mytarget.o test.c
```

The compiler generates dependency lines with the default target name `mytarget.o` instead of `test.o`.

Related information

C compiler option **--preprocess=+make** (Generate dependencies for make)

C compiler option **--dep-file** (Generate dependencies in a file)

C compiler option: --max-call-depth

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. In the **Maximum call depth for code compaction** field, enter a value.

Command line syntax

--max-call-depth=*value*

Default: -1

Description

This option is related to the compiler optimization **--optimize=+compact** (Code compaction or reverse inlining). Code compaction is the opposite of inlining functions: large sequences of code that occur more than once, are transformed into a function. This reduces code size (possibly at the cost of execution speed).

During code compaction it is possible that the compiler generates nested calls. This may cause the program to run out of its stack. To prevent stack overflow caused by too deeply nested function calls, you can use this option to limit the call depth. This option can have the following values:

-1   Poses no limit to the call depth (default)

0   The compiler will not generate any function calls. (Effectively the same as if you turned of code compaction with option **--optimize=-compact**)

> 0   Code sequences are only reversed if this will not lead to code at a call depth larger than specified with *value*. Function calls will be placed at a call depth no larger than *value*-1. (Note that if you specified a value of 1, the option **--optimize=+compact** may remain without effect when code sequences for reversing contain function calls.)

This option does not influence the call depth of user written functions.

If you use this option with various C modules, the call depth is valid for each individual module. The call depth after linking may differ, depending on the nature of the modules.

Related information

C compiler option **--optimize=+compact** (Optimization: code compaction)

C compiler option **--compact-max-size** (Maximum size of a match for code compaction)

C compiler option: --mil / --mil-split

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. Enable the option **Build for application wide optimizations (MIL linking)**.

3. Select **Optimize less/Build faster** or **Optimize more/Build slower**.

Command line syntax

```
--mil
--mil-split[=file,...]
```

Description

With option **--mil** the C compiler skips the code generator phase and writes the optimized intermediate representation (MIL) to a file with the suffix `.mil`. The C compiler accepts `.mil` files as input files on the command line.

Option **--mil-split** does the same as option **--mil**, but in addition, the C compiler splits the MIL representation and writes it to separate files with suffix `.ms`. One file is written for each input file or MIL library specified on the command line. The `.ms` files are only updated on a change. The C compiler accepts `.ms` files as input files on the command line.

With option **--mil-split** you can perform application-wide optimizations during the frontend phase by specifying all modules at once, and still invoke the backend phase one module at a time to reduce the total compilation time. Application wide code compaction is not possible in this case.

Optionally, you can specify another filename for the `.ms` file the C compiler generates. Without an argument, the basename of the C source file is used to create the `.ms` filename. Note that if you specify a filename, you have to specify one filename for every input file.

Note that with both options some extra strict type checking is done that can cause building to fail in a way that is unforeseen and difficult to understand. For example, when you use one of these options in combination with option **--uchar** and you link the MIL library, you might get the following error:

```
ctc E289: ["..\..\..\strlen.c" 14/1] "strlen" redeclared with a different type
ctc I802: ["installation-dir\ctc\include\string.h" 44/17]
          previous declaration of "strlen"
1 errors, 0 warnings
```

This is caused by the fact that the MIL library is built without **--uchar**. You can workaround this problem by rebuilding the MIL libraries.

Build for application wide optimizations (MIL linking) and Optimize less/Build faster

This option is standard MIL linking and splitting. Note that you can control the optimizations to be performed with the optimization settings.

Optimize more/Build slower

When you enable this option, the compiler's frontend does not split the MIL stream in separate modules, but feeds it directly to the compiler's backend, allowing the code compaction to be performed application wide.

Related information

Section 4.1, *Compilation Process*

Control program option **--mil-link** / **--mil-split**

C compiler option: --misrac

Menu entry

1. Select **C/C++ Compiler » MISRA C**.

2. Make a selection from the **MISRA C checking** list.

3. If you selected **Custom**, expand the **Custom 1998**, **Custom 2004** or **Custom 2012** entry and enable one or more individual rules.

Command line syntax

--misrac={**all** | *nr*[**-***nr*]},...

Description

With this option you specify to the compiler which MISRA C rules must be checked. With the option **--misrac=all** the compiler checks for all supported MISRA C rules.

Example

```
ctc --misrac=9-13 test.c
```

The compiler generates an error for each MISRA C rule 9, 10, 11, 12 or 13 violation in file `test.c`.

Related information

Section 4.7.2, *C Code Checking: MISRA C*

C compiler option **--misrac-mandatory-warnings**

C compiler option **--misrac-advisory-warnings**

C compiler option **--misrac-required-warnings**

Linker option **--misrac-report**

C compiler option: --misrac-advisory-warnings / --misrac-required-warnings / --misrac-mandatory-warnings

Menu entry

1. Select **C/C++ Compiler » MISRA C**.

2. Make a selection from the **MISRA C checking** list.

3. Enable one or more of the options:
   Warnings instead of errors for mandatory rules
   Warnings instead of errors for required rules
   Warnings instead of errors for advisory rules.

Command line syntax

```
--misrac-advisory-warnings
--misrac-required-warnings
--misrac-mandatory-warnings
```

Description

Normally, if an advisory rule or required rule is violated, the compiler generates an error. As a consequence, no output file is generated. With this option, the compiler generates a warning instead of an error.

Related information

Section 4.7.2, *C Code Checking: MISRA C*

C compiler option **--misrac**

Linker option **--misrac-report**

C compiler option: --misrac-version

Menu entry

1. Select **C/C++ Compiler » MISRA C**.

2. Select the **MISRA C version**: **1998**, **2004** or **2012**.

Command line syntax

`--misrac-version={1998|2004|2012}`

Default: 2004

Description

MISRA C rules exist in three versions: MISRA C:1998, MISRA C:2004 and MISRA C:2012. By default, the C source is checked against the MISRA C:2004 rules. With this option you can select which version to use.

Related information

Section 4.7.2, *C Code Checking: MISRA C*

C compiler option **--misrac**

C compiler option: --mmu-present / --mmu-on

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor Selection** list, select a processor that has an MMU or select a user defined processor.

3. (Optional) Select **C/C++ Compiler » Miscellaneous**.

4. (Optional) Add the option **--mmu-present** to the **Additional options** field.

5. Select **C/C++ Compiler » Code Generation**.

6. Enable the option **Use the MMU if present**.

> For the TriCore processor TC1130, the option **--mmu-present** is set automatically, so you only need to enable the option **Use the MMU if present**. For user defined processors you need to enable them both.

Command line syntax

```
--mmu-present
--mmu-on
```

Description

If the processor you are using has an MMU which is activated, you can tell the compiler to use the MMU. The compiler then will align data naturally. To instruct the compiler to use the MMU, you must set both the option **--mmu-present** and the option **--mmu-on**.

With the option **--mmu-present** you tell the compiler that an MMU is physically present. With the option **--mmu-on** you tell the compiler the MMU is activated. In this case the compiler needs to align data naturally. Both options are necessary.

If you select the TC1130 processor, the option **--mmu-present** is set automatically, based on the chosen target processor. If you are using a target processor with MMU that is not predefined, you need to set this option manually.

Note that the option **--mmu-on** is only required for the TC1130 when the MMU is used on that device. This option provides a workaround for silicon bug CPU_TC.052 Alignment Restrictions for Accesses using PTE-Based Translation, which states that natural alignment must be used for accesses undergoing PTE-based translation which may result in a non-cacheable memory access.

Example

To instruct the compiler to use the activated MMU:

```
ctc --mmu-present --mmu-on test.c
```

Related information

Control program option **--cpu** (Select processor)

C compiler option **--fp-model=+soft** (Disable hardware floating-point instructions)

C compiler option: --no-clear

Menu entry

1. Select **C Compiler » Allocation**.

2. Disable the option **Clear uninitialized global and static variables**.

Command line syntax

```
--no-clear
```

Description

Normally uninitialized global/static variables are cleared at program startup. With this option you tell the compiler to generate code to prevent uninitialized global/static variables from being cleared at program startup.

This option applies to constant as well as non-constant variables.

Related information

Pragmas `clear/noclear`

C compiler option: --no-stdinc

Menu entry

1.   Select **C/C++ Compiler » Miscellaneous**.

2.   Add the option **--no-stdinc** to the **Additional options** field.

Command line syntax

```
--no-stdinc
```

Description

With this option you tell the compiler not to look in the default `include` directory relative to the installation directory, when searching for include files. This way the compiler only searches in the include file search paths you specified.

Related information

C compiler option **--include-directory** (Add directory to include file search path)

Section 4.4, *How the Compiler Searches Include Files*

C compiler option: --no-warnings (-w)

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

   The Suppress C compiler warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.

3. Enter the numbers, separated by commas or as a range, of the warnings you want to suppress (for example `537,538`). Or you can use the **Add** button multiple times.

4. To suppress all warnings, enable the option **Suppress all warnings**.

Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

Command line syntax

--no-warnings[**=***number*[*-number*],...]

-w[*number*[*-number*],...]

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.

- If you specify this option but without numbers, all warnings are suppressed.

- If you specify this option with a number or a range, only the specified warnings are suppressed. You can specify the option **--no-warnings=***number* multiple times.

Example

To suppress warnings 537 and 538, enter:

```
ctc test.c --no-warnings=537,538
```

Related information

C compiler option **--warnings-as-errors** (Treat warnings as errors)

Pragma `warning`

C compiler option: --object-comment

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add your comment to the **Comment in object file** field.

Command line syntax

`--object-comment=`_comment_

Description

With this option the compiler generates a `.comment` section at the end of the assembly file. The section contains the _comment_ specified with this option. After assembling, this text is included in the `.o` object and `.elf` files. Place the comment between double quotes.

Instead of this option you can also specify the following pragma in your C source:

```
#pragma object_comment comment
```

Example

```
ctc --object-comment="Created by Altium" test.c
```

The compiler creates the file `test.src` with a `.comment` section at the end of the file. After assembling this file, the text "Created by Altium" is incorporated in the generated object file.

Related information

-

C compiler option: --optimize (-O)

Menu entry

1.   Select **C/C++ Compiler » Optimization**.

2.   Select an optimization level in the **Optimization level** box.

Command line syntax

--optimize[**=***flags*]

-O*flags*

You can set the following flags:

+/-coalesce	**a/A**	Coalescer: remove unnecessary moves
+/-cse	**c/C**	Common subexpression elimination
+/-expression	**e/E**	Expression simplification
+/-flow	**f/F**	Control flow simplification
+/-glo	**g/G**	Generic assembly code optimizations
+/-inline	**i/I**	Automatic function inlining
+/-schedule	**k/K**	Instruction scheduler
+/-loop	**l/L**	Loop transformations
+/-simd	**m/M**	Perform SIMD optimizations
+/-align-loop	**n/N**	Align loop bodies
+/-forward	**o/O**	Forward store
+/-propagate	**p/P**	Constant propagation
+/-compact	**r/R**	Code compaction (reverse inlining)
+/-subscript	**s/S**	Subscript strength reduction
+/-unroll	**u/U**	Unroll small loops
+/-ifconvert	**v/V**	Convert IF statements using predicates
+/-pipeline	**w/W**	Software pipelining
+/-peephole	**y/Y**	Peephole optimizations
+/-predict		Branch prediction

Use the following options for predefined sets of flags:

--optimize=0	**-O0**	No optimization Alias for **-OaCEFGIKLMNOPRSUVWY,-predict**

No optimizations are performed except for the coalescer (to allow better debug information). The compiler tries to achieve an optimal resemblance between source code and produced code. Expressions are

evaluated in the same order as written in the source code, associative and commutative properties are not used.

--optimize=1	**-O1**	Optimize
		Alias for **-OaCefgIKLMNOPRSUVWy,+predict**

Enables optimizations that do not affect the debug ability of the source code. Use this level when you encounter problems during debugging your source code with optimization level 2.

--optimize=2	**-O2**	Optimize more (default)
		Alias for **-OacefgIklMnoprsUvwy,+predict**

Enables more optimizations to reduce code size and/or execution time. This is the default optimization level.

--optimize=3	**-O3**	Optimize most
		Alias for **-Oacefgiklmnoprsuvwy,+predict**

This is the highest optimization level. Use this level to decrease execution time to meet your real-time requirements.

Default: **--optimize=2**

Description

With this option you can control the level of optimization. If you do not use this option, the default optimization level is *Optimize more* (option **--optimize=2** or **--optimize**).

When you use this option to specify a set of optimizations, you can overrule these settings in your C source file with #pragma optimize *flag* / #pragma endoptimize.

In addition to the option **--optimize**, you can specify the option **--tradeoff** (**-t**). With this option you specify whether the used optimizations should optimize for more speed (regardless of code size) or for smaller code size (regardless of speed).

Example

The following invocations are equivalent and result all in the default optimization set:

```
ctc test.c

ctc --optimize=2 test.c
ctc -O2 test.c

ctc --optimize test.c
ctc -O test.c

ctc -OacefgIklMNoprsUvwy test.c
ctc --optimize=+coalesce,+cse,+expression,+flow,+glo,-inline,
     +schedule,+loop,-simd,+align-loop,+forward,+propagate,
     +compact,+subscript,-unroll,+ifconvert,+pipeline,+peephole test.c
```

Related information

C compiler option **--tradeoff** (Trade off between speed and size)

Pragma `optimize/endoptimize`

Section 4.6, *Compiler Optimizations*

C compiler option: --option-file (-f)

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--option-file** to the **Additional options** field.

    Be aware that the options in the option file are added to the C compiler options you have set in the other pages. Only in extraordinary cases you may want to use them in combination.

Command line syntax

--option-file=*file*,...

-f *file*,...

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the compiler.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** multiple times.

Format of an option file

*   Multiple arguments on one line in the option file are allowed.

*   To include whitespace in an argument, surround the argument with single or double quotes.

*   If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

    ```
    "This has a single quote ' embedded"

    'This has a double quote " embedded'

    'This has a double quote " and a single quote '"' embedded"
    ```

*   When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

    ```
    "This is a continuation \
    line"

            -> "This is a continuation line"
    ```

- It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
--debug-info
--define=DEMO=1
test.c
```

Specify the option file to the compiler:

```
ctc --option-file=myoptions
```

This is equivalent to the following command line:

```
ctc --debug-info --define=DEMO=1 test.c
```

Related information

-

C compiler option: --output (-o)

Menu entry

Eclipse names the output file always after the C source file.

Command line syntax

--output=*file*

-o *file*

Description

With this option you can specify another filename for the output file of the compiler. Without this option the basename of the C source file is used with extension .src.

Example

To create the file output.src instead of test.src, enter:

```
ctc --output=output.src test.c
```

Related information

-

C compiler option: --preprocess (-E)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

2. Enable the option **Store preprocessor output in <file>.pre**.

3. (Optional) Enable the option **Keep comments in preprocessor output**.

4. (Optional) Enable the option **Keep #line info in preprocessor output**.

Command line syntax

--preprocess[**=***flags*]

-E[*flags*]

You can set the following flags:

+/-comments	**c/C**	keep comments
+/-includes	**i/I**	generate a list of included source files
+/-list	**l/L**	generate a list of macro definitions
+/-make	**m/M**	generate dependencies for make
+/-noline	**p/P**	strip #line source position information

Default: **-ECILMP**

Description

With this option you tell the compiler to preprocess the C source. Under Eclipse the compiler sends the preprocessed output to the file *name*.pre (where *name* is the name of the C source file to compile). Eclipse also compiles the C source.

On the command line, the compiler sends the preprocessed file to stdout. To capture the information in a file, specify an output file with the option **--output**.

With **--preprocess=+comments** you tell the preprocessor to keep the comments from the C source file in the preprocessed output.

With **--preprocess=+includes** the compiler will generate a list of all included source files. The preprocessor output is discarded.

With **--preprocess=+list** the compiler will generate a list of all macro definitions. The preprocessor output is discarded.

With **--preprocess=+make** the compiler will generate dependency lines that can be used in a Makefile. The preprocessor output is discarded. The default target name is the basename of the input file, with the extension .o. With the option **--make-target** you can specify a target name which overrules the default target name.

With **--preprocess=+noline** you tell the preprocessor to strip the #line source position information (lines starting with `#line`). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is easier to read.

Example

```
ctc --preprocess=+comments,-make,-noline test.c --output=test.pre
```

The compiler preprocesses the file `test.c` and sends the output to the file `test.pre`. Comments are included but no dependencies are generated and the line source position information is not stripped from the output file.

Related information

C compiler option **--dep-file** (Generate dependencies in a file)

C compiler option **--make-target** (Specify target name for **-Em** output)

C compiler option: --profile (-p)

Menu entry

1. Select **C/C++ Compiler » Debugging**.

2. Enable or disable **Static profiling**.

3. Enable or disable one or more of the following **Generate profiling information** options (dynamic profiling):

   - **for block counters** (not in combination with Call graph or Function timers)

   - **to build a call graph**

   - **for function counters**

   - **for function timers**

4. Optionally, when you selected function timers, enable option **Exclude time spent in interrupt functions**.

Note that the more detailed information you request, the larger the overhead in terms of execution time, code size and heap space needed. The option **--debug** does not affect profiling, execution time or code size.

Command line syntax

--profile[**=***flag*,...]

-p[*flags*]

Use the following option for a predefined set of flags:

--profile=g	**-pg**	Profiling with call graph and function timers. Alias for: **-pBcFStI**

You can set the following flags:

+/-block	**b/B**	block counters
+/-callgraph	**c/C**	call graph
+/-function	**f/F**	function counters
+/-interrupt	**i/I**	exclude interrupt time
+/-static	**s/S**	static profile generation
+/-time	**t/T**	function timers

Default: **-pBCFSTI**

Default (without flags): **-pBCfSTI**

Description

Profiling is the process of collecting statistical data about a running application. With these data you can analyze which functions are called, how often they are called and what their execution time is.

Several methods of profiling exist. One method is *code instrumentation* which adds code to your application that takes care of the profiling process when the application is executed. Another method is *static profiling*.

For an extensive description of profiling refer to Chapter 12, *Profiling*.

You can obtain the following profiling data (see flags above):

Block counters (not in combination with Call graph or Function timers)

This will instrument the code to perform basic block counting. As the program runs, it counts the number of executions of each branch in an if statement, each iteration of a for loop, and so on. Note that though you can combine Block counters with Function counters, this has no effect because Function counters is only a subset of Block counters.

Call graph (not in combination with Block counters)

This will instrument the code to reconstruct the run-time call graph. As the program runs it associates the caller with the gathered profiling data.

Function counters

This will instrument the code to perform function call counting. This is a subset of the basic Block counters.

Function timers (not in combination with Block counters/Function counters)

This will instrument the code to measure the time spent in a function. This includes the time spent in all sub functions (callees). Also the time spent in interrupt functions is taken into account, unless you enable option **Exclude time spent in interrupt functions** (**-pti**).

Static profiling

With this option you do not need to run the application to get profiling results. The compiler generates profiling information at compile time, without adding extra code to your application.

If you use one or more profiling options that use code instrumentation, you must link the corresponding libraries too! Refer to Section 7.3, *Linking with Libraries*, for an overview of the (profiling) libraries. In Eclipse the correct libraries are linked automatically.

Example

To generate block count information for the module `test.c` during execution, compile as follows:

```
ctc --profile=+block test.c
```

In this case you must link the library `libpb.a`.

Related information

Chapter 12, *Profiling*

C compiler option: --relax-compact-name-check

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--relax-compact-name-check** to the **Additional options** field.

Command line syntax

```
--relax-compact-name-check
```

Description

With code compaction (reverse inlining), chunks of code that can occur more than once in different functions, are transformed into another function. By default, the compiler only performs code compaction on sections that have the same section type prefix, core association and name given by the section renaming pragma or option. Chunks of code that are part of functions with a different core association or section rename suffix are not taken into account. With this option the compiler does not perform this section name check, but performs code compaction whenever possible.

Related information

Section 4.6.2, *Core Specific Optimizations (backend)*

C compiler option: --relax-overlay-name-check

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--relax-overlay-name-check** to the **Additional options** field.

Command line syntax

```
--relax-overlay-name-check
```

Description

This option relaxes the overlaying of romdata for internal constants, string literals and compound literals. Romdata for internals are overlaid when equal. By default, the compiler only performs overlaying on equal romdata for internals that have the same memory space, core association and section rename suffix. With this option the compiler does not perform this overlay name check, but performs overlaying whenever possible.

Related information

-

C compiler option: --rename-sections (-R)

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--rename-sections** to the **Additional options** field.

Command line syntax

--rename-sections[**=**name]

-R[name]

Description

The compiler defaults to a section naming convention, using a prefix indicating the section type, the module name and a symbol name:

section_type_prefix.module_name.symbol_name

For example, .text.*module_name.symbol_name* for code sections.

In case a module must be loaded at a fixed address, or a data section needs a special place in memory, you can use this option to generate different section names (*section_type_prefix.name* where *name* replaces the part *module_name.symbol_name*). You can then use this unique section name in the linker script file for locating.

If you use this option without a value, the compiler uses the default section naming.

Example

To generate the section name *section_type_prefix*.NEW instead of the default section name *section_type_prefix.module_name.symbol_name*, enter:

ctc -RNEW test.c

To generate the section name *section_type_prefix* instead of the default section name *section_type_prefix.module_name.symbol_name*, enter:

ctc -R" " test.c   (note the single space between the quotes)

Related information

Section 1.12, *Compiler Generated Sections*

C compiler option: --runtime (-r)

Menu entry

1. Select **C/C++ Compiler » Debugging**.

2. Enable or disable one or more of the following run-time error checking options:

    - Generate code for bounds checking

    - Generate code to detect unhandled case in a switch

    - Generate code for malloc consistency checks

Command line syntax

--runtime[**=***flag*,...]

-r[*flags*]

You can set the following flags:

+/-bounds	**b/B**	bounds checking
+/-case	**c/C**	report unhandled case in a switch
+/-malloc	**m/M**	malloc consistency checks

Default (without flags): **-rbcm**

Description

This option controls a number of run-time checks to detect errors during program execution. Some of these checks require additional code to be inserted in the generated code, and may therefore slow down the program execution. The following checks are available:

Bounds checking

Every pointer update and dereference will be checked to detect out-of-bounds accesses, null pointers and uninitialized automatic pointer variables. This check will increase the code size and slow down the program considerably. In addition, some heap memory is allocated to store the bounds information. You may enable bounds checking for individual modules or even parts of modules only (see `#pragma runtime`).

Report unhandled case in a switch

Report an unhandled case value in a switch without a default part. This check will add one function call to every switch without a default part, but it will have little impact on the execution speed.

Malloc consistency checks

This option enables the use of wrappers around the functions malloc/realloc/free that will check for common dynamic memory allocation errors like:

• buffer overflow

• write to freed memory

• multiple calls to free

• passing invalid pointer to free

Enabling this check will extract some additional code from the library, but it will not enlarge your application code. The dynamic memory usage will increase by a couple of bytes per allocation.

Related information

Pragma `runtime`

C compiler option: --section-name-with-symbol

Menu entry

1.   Select **C/C++ Compiler » Miscellaneous**.

2.   Add the option **--section-name-with-symbol** to the **Additional options** field.

Command line syntax

```
--section-name-with-symbol
```

Description

Normally, when you use an option or pragma to influence section naming, the symbol name is not included.

With this option you tell the compiler to include the symbol name in the section name.

Instead of this option you can use the following pragma:

```
#pragma section_name_with_symbol
```

Related information

Section 1.12.1, *Rename Sections*

C compiler option: --signed-bitfields

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat 'int' bit-fields as signed**.

Command line syntax

`--signed-bitfields`

Description

For bit-fields it depends on the implementation whether a plain `int` is treated as `signed int` or `unsigned int`. By default an `int` bit-field is treated as `unsigned int`. This offers the best performance. With this option you tell the compiler to treat `int` bit-fields as `signed int`. In this case, you can still add the keyword `unsigned` to treat a particular `int` bit-field as `unsigned`.

Related information

C++ compiler option **--signed-bitfields**

Section 1.1, *Data Types*

C compiler option: --silicon-bug

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor selection** list, select a processor.

   *The **CPU problem bypasses and checks** box shows the available workarounds/checks available for the selected processor.*

3. (Optional) Select **Show all CPU problem bypasses and checks**.

4. Click **Select All** or select one or more individual options.

Command line syntax

```
--silicon-bug=arg,...
```

You can give one or more of the following arguments:

all-tc1130	All TriCore TC1130 workarounds
all-tc1164	All TriCore TC1164 workarounds
all-tc1166	All TriCore TC1166 workarounds
all-tc131	All TriCore 1.3.1 workarounds
all-tc1736	All TriCore TC1736 workarounds
all-tc1762	All TriCore TC1762 workarounds
all-tc1764	All TriCore TC1764 workarounds
all-tc1766	All TriCore TC1766 workarounds
all-tc1767	All TriCore TC1767 workarounds
all-tc1792	All TriCore TC1792 workarounds
all-tc1796	All TriCore TC1796 workarounds
all-tc1797	All TriCore TC1797 workarounds
cpu-tc013	Workaround for CPU_TC.013
cpu-tc048	Workaround for CPU_TC.048
cpu-tc060	Workaround for CPU_TC.060
cpu-tc065	Workaround for CPU_TC.065
cpu-tc068	Workaround for CPU_TC.068
cpu-tc069	Workaround for CPU_TC.069
cpu-tc070	Workaround for CPU_TC.070
cpu-tc072	Workaround for CPU_TC.072
cpu-tc083	Workaround for CPU_TC.083

cpu-tc094	Workaround for CPU_TC.094
cpu-tc095	Workaround for CPU_TC.095
cpu-tc096	Workaround for CPU_TC.096
cpu-tc103	Workaround for CPU_TC.103
cpu-tc104	Workaround for CPU_TC.104
cpu-tc105	Check for CPU_TC.105
cpu-tc106	Check for CPU_TC.106
cpu-tc108	Workaround for CPU_TC.108
cpu-tc109	Workaround for CPU_TC.109
cpu-tc116	Workaround for CPU_TC.116

Description

With this option you specify for which hardware problems the compiler should generate workarounds. Please refer to Chapter 18, *CPU Problem Bypasses and Checks* for more information about the individual problems and workarounds.

Instead of this option you can use the following pragma:

```
#pragma CPU_functional_problem
```

For example:

```
#pragma CPU_TC013
```

Example

To enable workarounds for problems CPU_TC.013 and CPU_TC.048, enter:

```
ctc --silicon-bug=cpu-tc013,cpu-tc048 test.c
```

Related information

Chapter 18, *CPU Problem Bypasses and Checks*

Assembler option **--silicon-bug**

C compiler option: --software-partitioning-info

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

2. Enable the option **Check software partitioning**.

Eclipse also calls the integrity check tool.

Command line syntax

`--software-partitioning-info`

Description

With this option the compiler adds information about symbol accesses to the output file in `.debug_tasking_verif` sections. This information can be used by the integrity check utility.

Example

```
ctc --software-partitioning-info test.c
```

Related information

Section 8.10, *Integrity Check Utility*

Section 8.6.2, *HLL Dump Output Format*

C compiler option: --source (-s)

Menu entry

1.   Select **C/C++ Compiler » Miscellaneous**.

2.   Enable the option **Merge C source code with generated assembly**.

Command line syntax

```
--source
```

```
-s
```

Description

With this option you tell the compiler to merge C source code with generated assembly code in the output file. The C source lines are included as comments.

Related information

Pragmas `source/nosource`

C compiler option: --static

Menu entry

-

Command line syntax

`--static`

Description

With this option, the compiler treats external definitions at file scope (except for `main`) as if they were declared `static`. As a result, unused functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module.

This option only makes sense when you specify all modules of an application on the command line.

To overrule this option for a specific function or variable, you can use the `export` attribute. For example, when a variable is accessed from assembly:

```
int i __attribute__((export)); /* 'i' has external linkage */
```

With the `export` attribute the compiler will not perform optimizations that affect the unknown code.

Example

```
ctc --static module1.c module2.c module3.c ...
```

Related information

-

C compiler option: --stdout (-n)

Menu entry

-

Command line syntax

--stdout

-n

Description

With this option you tell the compiler to send the output to stdout (usually your screen). No files are created. This option is for example useful to quickly inspect the output or to redirect the output to other tools.

Related information

-

C compiler option: --switch

Menu entry

1. Select **C/C++ Compiler » Code Generation**.

2. Select an **Algorithm for switch statements**.

Command line syntax

--switch==*arg*

You can give one of the following arguments:

auto	Choose most optimal code
jumptab	Generate jump tables
linear	Use linear jump chain code
lookup	Generate lookup tables

Default: **--switch=auto**

Description

With this option you tell the compiler which code must be generated for a switch statement: a jump chain (linear switch), a jump table or a lookup table. By default, the compiler will automatically choose the most efficient switch implementation based on code and data size and execution speed.

Instead of this option you can use the following pragma:

```
#pragma switch arg
```

Example

To use a table filled with target addresses for each possible switch value, enter:

```
ctc --switch=jumptab test.c
```

Related information

Section 1.10, *Switch Statement*

C compiler option: --tradeoff (-t)

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. Select a trade-off level in the **Trade-off between speed and size** box.

Command line syntax

--tradeoff={0|1|2|3|4}

-t{0|1|2|3|4}

Default: **--tradeoff=4**

Description

If the compiler uses certain optimizations (option **--optimize**), you can use this option to specify whether the used optimizations should optimize for more speed (regardless of code size) or for smaller code size (regardless of speed).

By default the compiler optimizes for code size (**--tradeoff=4**).

> If you have not specified the option **--optimize**, the compiler uses the default *Optimize more* optimization. In this case it is still useful to specify a trade-off level.
>
> With option **--tradeoff=3** or **--tradeoff=4** the loop alignment optimization is switched off.

Example

To set the trade-off level for the used optimizations:

```
ctc --tradeoff=2 test.c
```

The compiler uses the default *Optimize more* optimization level and balances speed and size while optimizing.

Related information

C compiler option **--optimize** (Specify optimization level)

Section 4.6.3, *Optimize for Code Size or Execution Speed*

C compiler option: --uchar (-u)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat 'char' variables as unsigned**.

Command line syntax

```
--uchar
```

```
-u
```

Description

By default `char` is the same as specifying `signed char`. With this option `char` is the same as `unsigned char`.

Note that this option can cause conflicts when you use it in combination with MIL linking. With MIL linking some extra strict type checking is done that can cause building to fail in a way that is unforeseen and difficult to understand. For example, when you use option **--mil** in combination with option **--uchar** and you link the MIL library, you might get the following error:

```
ctc E289: ["..\..\..\strlen.c" 14/1] "strlen" redeclared with a different type
ctc I802: ["installation-dir\ctc\include\string.h" 44/17]
          previous declaration of "strlen"
1 errors, 0 warnings
```

This is caused by the fact that the MIL library is built without **--uchar**. You can workaround this problem by rebuilding the MIL libraries.

Related information

Section 1.1, *Data Types*

C compiler option: --undefine (-U)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**

    The Defined symbols box shows the symbols that are currently defined.

2. To remove a defined symbol, select the symbol in the **Defined symbols** box and click on the **Delete** button.

Command line syntax

`--undefine=`*macro_name*

`-U`*macro_name*

Description

With this option you can undefine an earlier defined macro as with `#undef`. This option is for example useful to undefine predefined macros.

The following predefined ISO C standard macros cannot be undefined:

`__FILE__`	current source filename
`__LINE__`	current source line number (int type)
`__TIME__`	hh:mm:ss
`__DATE__`	Mmm dd yyyy
`__STDC__`	level of ANSI standard

Example

To undefine the predefined macro `__TASKING__`:

`ctc --undefine=__TASKING__ test.c`

Related information

C compiler option **--define** (Define preprocessor macro)

Section 1.9, *Predefined Preprocessor Macros*

C compiler option: --unroll-factor

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--unroll-factor** to the **Additional options** field.

Command line syntax

--unroll-factor=*value*

Default: **--unroll-factor=-1**

Description

With the loop unrolling optimization, short loops are eliminated by replacing them with a number of copies to reduce the number of branches. With this option you specify how many times eligible loops should be unrolled. When the unroll factor is -1 (default), small loops are unrolled automatically if the loop unrolling optimization (**--optimize=+unroll / -Ou**) is enabled and the optimization trade-off is set for speed (**--tradeoff=0 / -t0**)).

Instead of this option you can use the following pragmas:

```
#pragma unroll_factor value
  ...
#pragma endunroll_factor
```

Example

To allow an unroll factor of four, enter:

```
ctc --optimize=+unroll --unroll-factor=4 --tradeoff=0 test.c
```

Related information

Pragma `unroll_factor`

C compiler option **--optimize** (Specify optimization level)

C compiler option **--tradeoff** (Trade off between speed and size)

Section 4.6, *Compiler Optimizations*

C compiler option: --user-mode

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--user-mode** to the **Additional options** field.

Command line syntax

`--user-mode=`*mode*

You can specify the following *mode* arguments:

user-0	User-0 unprivileged mode
user-1	User-1 privileged mode
kernel	Kernel/Supervisor mode

Default: `--user-mode=kernel`

Description

With this option you specify the mode (I/O privilege mode) the TriCore runs in: User-0, User-1 or Kernel/Supervisor. The availability of some instructions depends on this mode. Most instructions run in all modes. The instructions enable and disable run in User-1 or kernel mode only. The instructions bisr, mtcr, cachea.i and tlb instructions run in kernel mode only.

Instead of this option you can use the following pragma:

#pragma user_mode *mode*

Example

To restrict the instructions in the assembly code to User-1 mode, enter:

ctc --user-mode=user-1 test.c

Related information

Pragma user_mode

Assembler option **--user-mode** (Select user mode)

C compiler option: --verbose (-v)

Menu entry

-

Command line syntax

`--verbose`

`-v`

Description

With this option you put the compiler in verbose mode. With this option the C compiler can generate additional informational diagnostics when available. For example, when code is generated to circumvent functional problems in the processor.

Related information

-

C compiler option: --version (-V)

Menu entry

-

Command line syntax

--version

-V

Description

Display version information. The compiler ignores all other options or input files.

Example

```
ctc --version
```

The compiler does not compile any files but displays the following version information:

```
TASKING VX-toolset for TriCore: C compiler   vx.yrz Build nnn
Copyright 2002-year Altium BV                 Serial# 00000000
```

Related information

-

C compiler option: --warnings-as-errors

Menu entry

1. Select **Global Options**.

2. Enable the option **Treat warnings as errors**.

Command line syntax

`--warnings-as-errors``[=number[-number],...]`

Description

If the compiler encounters an error, it stops compiling. When you use this option without arguments, you tell the compiler to treat all warnings not suppressed by option **--no-warnings** (or `#pragma warning`) as errors. This means that the exit status of the compiler will be non-zero after one or more compiler warnings. As a consequence, the compiler now also stops after encountering a warning.

You can limit this option to specific warnings by specifying a comma-separated list of warning numbers or ranges. In this case, this option takes precedence over option **--no-warnings** (and `#pragma warning`).

Related information

C compiler option **--no-warnings** (Suppress some or all warnings)

Pragma `warning`

10.3. C++ Compiler Options

This section lists all C++ compiler options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the C++ compiler via the control program. Therefore, it uses the syntax of the control program to pass options and files to the C++ compiler. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

1. From the **Project** menu, select **Properties for**

   The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » Miscellaneous**.

4. In the **Additional options** field, enter one or more command line options.

   *Because Eclipse uses the control program, you have to precede the option with **-Wcp** to pass the option via the control program directly to the C++ compiler.*

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

If an option requires an argument, the argument may be separated from the keyword by white space, or the keyword may be immediately followed by **=**option. When the second form is used there may not be any white space on either side of the equal sign.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+**longflag. To switch a flag off, use an uppercase letter or a **-**longflag. Separate longflags with commas. The following two invocations are equivalent:

```
cptc -Ecp test.cc
cptc --preprocess=+comments,+noline test.cc
```

When you do not specify an option, a default value may become active.

The priority of the options is left-to-right: when two options conflict, the first (most left) one takes effect. The **-D** and **-U** options are not considered conflicting options, so they are processed left-to-right for each source file. You can overrule the default output file name with the **--output-file** option.

C++ compiler option: --alternative-tokens

Menu entry

-

Command line syntax

`--alternative-tokens`

Description

Enable recognition of alternative tokens. This controls recognition of the digraph tokens in C++, and controls recognition of the operator keywords (e.g., `not`, `and`, `bitand`, etc.).

Example

To enable operator keywords (e.g., "not", "and") and digraphs, enter:

```
cptc --alternative-tokens test.cc
```

Related information

-

C++ compiler option: --anachronisms

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **C++ anachronisms**.

Command line syntax

```
--anachronisms
```

Description

Enable C++ anachronisms. This option also enables **--nonconst-ref-anachronism**. But you can turn this off individually with option **--no-nonconst-ref-anachronism**.

Related information

C++ compiler option **--nonconst-ref-anachronism** (Nonconst reference anachronism)

Section 2.2.3, *Anachronisms Accepted*

C++ compiler option: --auto-type

Menu entry

-

Command line syntax

```
--auto-type
```

```
--no-auto-type
```

Default: `auto` is a storage class specifier.

Description

Enable or disable `auto` as a type specifier where the actual type is deduced from an initializer that follows.

This feature is implicitly enabled in C++11 mode.

Related information

C++ compiler option **--no-auto-storage** (Do not allow `auto` as storage specifier)

C++ compiler option **--c++11** (C++11 language extensions)

C++ compiler option: --base-assign-op-is-default

Menu entry

-

Command line syntax

`--base-assign-op-is-default`

Description

Enable the anachronism of accepting a copy assignment operator that has an input parameter that is a reference to a base class as a default copy assignment operator for the derived class.

Related information

-

C++ compiler option: --building-runtime

Menu entry

-

Command line syntax

`--building-runtime`

Description

Special option for building the C++ run-time library. Used to indicate that the C++ run-time library is being compiled. This causes additional macros to be predefined that are used to pass configuration information from the C++ compiler to the run-time.

Related information

-

C++ compiler option: --c++03

Menu entry

1.   Select **C/C++ Compiler » Language**.

2.   From the **Comply to C++ standard** list, select **ISO C++03**.

Command line syntax

`--c++03`

Description

Enable the C++ extensions that are defined by the ISO/IEC 14882:2003 standard. C++11 extensions are disabled, unless you enable individual options.

Related information

-

C++ compiler option: --c++11

Menu entry

1. Select **C/C++ Compiler » Language**.

2. From the **Comply to C++ standard** list, select **ISO C++11**.

Command line syntax

`--c++11`

Description

Enable the C++ extensions that are defined by the C++11 standard (ISO/IEC 14882:2011). This option also enables exceptions.

Related information

C++ compiler option **--exceptions** (Enable exceptions)

C++ compiler option: --c++11-sfinae

Menu entry

-

Command line syntax

`--c++11-sfinae`

`--no-c++11-sfinae`

Description

Enable or disable template deduction in the style dictated by the C++11 standard (SFINAE rules of document N2634), i.e., where general expressions are allowed in deduction contexts and they undergo the full usual semantic checking. This type of deduction is necessary to get the full power of the `decltype` feature in return types. "SFINAE" refers to the initials of the phrase "Substitution Failure Is Not An Error", which is the guiding principle for template deduction, and by extension a name for the process of deduction.

This feature is implicitly enabled in C++11 mode (option **--c++11**), and is implicitly disabled in GNU modes.

Related information

C++ compiler option **--c++11** (C++11 language extensions)

C++ compiler option: --c++11-sfinae-ignore-access

Menu entry

-

Command line syntax

`--c++11-sfinae-ignore-access`

`--no-c++11-sfinae-ignore-access`

Default: `--no-c++11-sfinae-ignore-access`

Description

When C++11 SFINAE is enabled (option **--c++11-sfinae**), option **--c++11-sfinae-ignore-access** indicates that access errors are not counted as errors that cause deduction failures. In document N2634 access errors are ignored, but the standards committee changed its mind about that later. So, the default (option **--no-c++11-sfinae-ignore-access**) is that access errors cause a deduction failure.

Related information

C++ compiler option **--c++11-sfinae** (C++11 SFINAE rules)

C++ compiler option: --c++14

Menu entry

1.   Select **C/C++ Compiler » Language**.

2.   From the **Comply to C++ standard** list, select **ISO C++14**.

Command line syntax

`--c++14`

Description

Enable the C++ extensions that are defined by the C++14 standard (ISO/IEC 14882:2014). This option also enables exceptions.

Related information

C++ compiler option **--exceptions** (Enable exceptions)

C++ compiler option: --check

Menu entry

-

Command line syntax

`--check`

Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application because the code will not actually be compiled.

The C++ compiler reports any warnings and/or errors.

This option is available on the command line only.

Related information

C compiler option **--check** (Check syntax)

Assembler option **--check** (Check syntax)

C++ compiler option: --check-concatenations

Menu entry

-

Command line syntax

`--check-concatenations`

Description

With this option the preprocessor will generate a diagnostic when a macro concatenation (such as `a ## b`) does not result in a valid token.

Related information

-

C++ compiler option: --compound-literals

Menu entry

-

Command line syntax

`--compound-literals`

Description

Allow compound literals in expressions. A compound literal looks like a cast containing an initializer. Its value is an object of the type specified in the cast, containing the elements specified in the initializer; it is an lvalue.

For example:

```
static int i[] = (int []) {1, 2, 3};
```

Related information

-

C++ compiler option: --context-limit

Menu entry

-

Command line syntax

`--context-limit=`*number*

Default: `--context-limit=10`

Description

Set the context limit to *number*. The context limit is the maximum number of template instantiation context entries to be displayed as part of a diagnostic message. If the number of context entries exceeds the limit, the first and last *N* context entries are displayed, where *N* is half of the context limit. A value of zero is used to indicate that there is no limit.

Example

To set the context limit to 5, enter:

```
cptc --context-limit=5 test.cc
```

Related information

-

C++ compiler option: --core

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor Selection** list, select a processor.

Command line syntax

--core=*core*

You can specify the following *core* arguments:

tc1.3	TriCore 1.3 architecture
tc1.3.1	TriCore 1.3.1 architecture
tc1.6	TriCore 1.6 architecture
tc1.6.x	TriCore 1.6.x architecture
tc1.6.2	TriCore 1.6.2 architecture

Default: **tc1.3**

Description

With this option you specify the core architecture for a target processor for which you create your application. If you use Eclipse or the control program, the TriCore toolset derives the core from the processor you selected.

The C++ compiler defines the macro __CORE_*core*__ depending on the *core*. The *core* is converted to uppercase and '.' is removed. For example, if you specify**--core=tc1.3.1**, the C++ compiler defines the macro __CORE_TC131__. When you do not specify **--core**, the C++ compiler defines __CORE_TC13__.

Example

Specify a custom core:

```
cptc --core=tc1.3.1 test.cc
```

Related information

Control program option **--cpu** (Select processor)

C++ compiler option: --create-pch

Menu entry

1. Select **C/C++ Compiler » Precompiled C++ Headers**.

2. Enter a filename in the **Create precompiled header file** field.

Command line syntax

--create-pch=_filename_

Description

If other conditions are satisfied, create a precompiled header file with the specified name. If **--pch** (automatic PCH mode) or **--use-pch** appears on the command line following this option, its effect is erased.

Example

To create a precompiled header file with the name test.pch, enter:

```
cptc --create-pch=test.pch test.cc
```

Related information

C++ compiler option **--pch** (Automatic PCH mode)

C++ compiler option **--use-pch** (Use precompiled header file)

Section 2.9, *Precompiled Headers*

C++ compiler option: --default-nocommon-tentative-definitions

Menu entry

-

Command line syntax

`--default-nocommon-tentative-definitions`

Description

In GNU C++ mode, this option prevents tentative definitions to be placed in common storage.

Related information

-

C++ compiler option: --defer-parse-function-templates

Menu entry

-

Command line syntax

`--defer-parse-function-templates`

`--no-defer-parse-function-templates`

Default: `--defer-parse-function-templates` in GNU C++ mode.

Description

Enable or disable deferral of prototype instantiations until the first actual instantiation of a function. This is used to permit the compilation of programs that contain definitions of unusable function templates.

It is enabled by default in GNU C++ mode.

Related information

-

C++ compiler option: --define (-D)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

    The Defined symbols box shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, `demo=1`)

Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

`--**define**=macro_name[(parm-list)][=macro_definition]`

`-Dmacro_name(parm-list)][=macro_definition]`

Description

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

Function-style macros can be defined by appending a macro parameter list to *macro_name*.

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, you can use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the C++ compiler with the option **--option-file** (**-f**) *file*.

The definition can be tested by the preprocessor with `#if`, `#ifdef` and `#ifndef`, for conditional compilations.

Make sure you do not use a reserved keyword as a macro name, as this can lead to unexpected results.

Example

Consider the following program with conditional code to compile a demo program and a real program:

```
void main( void )
{
#if DEMO
    demo_func();   /* compile for the demo program */
#else
    real_func();   /* compile for the real program */
```

```
#endif
}
```

You can now use a macro definition to set the DEMO flag:

```
cptc --define=DEMO test.cc
cptc --define=DEMO=1 test.cc
```

Note that both invocations have the same effect.

The next example shows how to define a macro with arguments. Note that the macro name and definition are placed between double quotes because otherwise the spaces would indicate a new option.

```
cptc --define="MAX(A,B)=((A) > (B) ? (A) : (B))" test.cc
```

Related information

C++ compiler option **--undefine** (Remove preprocessor macro)

C++ compiler option **--option-file** (Specify an option file)

C++ compiler option: --delegating-constructors

Menu entry

-

Command line syntax

`--delegating-constructors`

Description

Enable delegating constructors (a C++11 feature).

Related information

-

C++ compiler option: --dep-file

Menu entry

-

Command line syntax

--dep-file[**=***file*]

Description

With this option you tell the C++ compiler to generate dependency lines that can be used in a Makefile. In contrast to the option **--preprocess=+make**, the dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension `.d` (one for every input file). When you specify a filename, all dependencies will be combined in the specified file.

Example

```
cptc --dep-file=test.dep test.cc
```

The C++ compiler compiles the file `test.cc`, which results in the output file `test.ic`, and generates dependency lines in the file `test.dep`.

Related information

C++ compiler option **--preprocess=+make** (Generate dependencies for make)

C++ compiler option: --deprecated-string-conv

Menu entry

-

Command line syntax

`--deprecated-string-conv`

Description

With this option the deprecated conversion from a string literal to `char *` is allowed. This is only meaningful when string literals are `const`.

Related information

-

C++ compiler option: --diag

Menu entry

-

Command line syntax

`--diag`

Description

With this option the C++ compiler displays a list of all diagnostic messages on stdout (usually your screen). The C++ compiler does not compile any files.

Related information

Section 5.3, *C++ Compiler Error Messages*

C++ compiler option: --dollar

Menu entry

-

Command line syntax

`--dollar`

Default format: No dollar signs are allowed in identifiers.

Description

Accept dollar signs in identifiers. Names like A$VAR are allowed.

Related information

-

C++ compiler option: --embedded-c++

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Comply to embedded C++ subset**.

Command line syntax

`--embedded-c++`

Description

The "Embedded C++" subset does not support templates, exceptions, namespaces, new-style casts, RTTI, multiple inheritance, virtual base classes, and the `mutable` keyword. Select this option when you want the C++ compiler to give an error when you use any of them in your C++ source.

Related information

-

C++ compiler option: --error-file

Menu entry

-

Command line syntax

--error-file[**=***file*]

Description

With this option the C++ compiler redirects error messages to a file. If you do not specify a filename, the error file will be named after the output file with extension .ecp.

Example

To write errors to errors.ecp instead of stderr, enter:

```
cptc --error-file=errors.ecp test.cc
```

Related information

-

C++ compiler option: --error-limit (-e)

Menu entry

-

Command line syntax

`--error-limit=``number`

`-e``number`

Default: **`--error-limit=100`**

Description

Set the error limit to *number*. The C++ compiler will abandon compilation after this number of errors (remarks and warnings are not counted). By default, the limit is 100.

Example

When you want compilation to stop when 10 errors occurred, enter:

```
cptc --error-limit=10 test.cc
```

Related information

-

C++ compiler option: --exceptions (-x)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable or disable the option **Support for C++ exception handling**.

Command line syntax

```
--exceptions
-x
```

```
--no-exceptions
```

Default: **--no-exceptions**

Default in C++11/C++14 mode: **--exceptions**

Description

With this option you enable or disable support for exception handling in the C++ compiler.

The macro __EXCEPTIONS is defined when exception handling support is enabled.

This feature is implicitly enabled in C++11 mode and C++14 mode.

Related information

C++ compiler option **--c++11** (C++11 language extensions)

C++ compiler option **--c++14** (C++14 language extensions)

C++ compiler option: --extended-variadic-macros

Menu entry

-

Command line syntax

`--extended-variadic-macros`

`--no-extended-variadic-macros`

Description

Enable or disable support for macros with a variable number of arguments (implies **--variadic-macros**) and allow the naming of the variable argument list.

Related information

C++ compiler option **--variadic-macros** (Allow variadic macros)

C++ compiler option: --force-vtbl

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Enable the option **Force definition of virtual function tables (C++)**.

Command line syntax

```
--force-vtbl
```

Description

Force definition of virtual function tables in cases where the heuristic used by the C++ compiler to decide on definition of virtual function tables provides no guidance.

Related information

C++ compiler option **--suppress-vtbl** (Suppress definition of virtual function tables)

C++ compiler option: --friend-injection

Menu entry

-

Command line syntax

--friend-injection

Default: `friend` names are not injected.

Description

Controls whether the name of a class or function that is declared only in `friend` declarations is visible when using the normal lookup mechanisms. When `friend` names are injected, they are visible to such lookups. When `friend` names are not injected (as required by the standard), function names are visible only when using argument-dependent lookup, and class names are never visible.

Related information

C++ compiler option **--no-arg-dep-lookup** (Disable argument dependent lookup)

C++ compiler option: --g++

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Allow GNU C++ extensions**.

Command line syntax

`--g++`

Description

Enable GNU C++ compiler language extensions.

Related information

Section 2.2.2, *GNU C++ Mode*

C++ compiler option: --generate-move-operations

Menu entry

-

Command line syntax

`--generate-move-operations`

Description

With this option the C++ compiler generates move constructors and move assignment operators according to standard C++11 rules.

Related information

-

C++ compiler option: --gnu-version

Menu entry

-

Command line syntax

`--gnu-version=``version`

Default: 30300 (version 3.3.0)

Description

It depends on the GNU C++ compiler version if a particular GNU extension is supported or not. With this option you set the GNU C++ compiler version that should be emulated in GNU C++ mode. Version *x.y.z* of the GNU C++ compiler is represented by the value $x*10000+y*100+z$.

Example

To specify version 3.4.1 of the GNU C++ compiler, enter:

```
cptc --g++ --gnu-version=30401 test.cc
```

Related information

Section 2.2.2, *GNU C++ Mode*

C++ compiler option: --guiding-decls

Menu entry

-

Command line syntax

--guiding-decls

Description

Enable recognition of "guiding declarations" of template functions. A guiding declaration is a function declaration that matches an instance of a function template but has no explicit definition (since its definition derives from the function template). For example:

```
template <class T> void f(T) { ... }
void f(int);
```

When regarded as a guiding declaration, `f(int)` is an instance of the template; otherwise, it is an independent function for which a definition must be supplied.

Related information

C++ compiler option **--old-specializations** (Old-style template specializations)

C++ compiler option: --help (-?)

Menu entry

-

Command line syntax

--help[**=**_item_]

-?

You can specify the following arguments:

options   Show extended option descriptions

Description

Displays an overview of all command line options. When you specify an argument you can list extended information such as a list of option descriptions.

Example

The following invocations all display a list of the available command line options:

```
cptc -?
cptc --help
cptc
```

The following invocation displays an extended list of the available options:

```
cptc --help=options
```

Related information

-

C++ compiler option: --ignore-std

Menu entry

-

Command line syntax

`--ignore-std`

Description

Enable a GNU C++ compatibility feature that makes the std namespace a synonym for the global namespace.

Related information

-

C++ compiler option: --implicit-extern-c-type-conversion

Menu entry

-

Command line syntax

`--implicit-extern-c-type-conversion`

Description

Enable the implicit type conversion between pointers to `extern "C"` and `extern "C++"` function types.

Related information

-

C++ compiler option: --implicit-include

Menu entry

1.   Select **C/C++ Compiler » Miscellaneous**.

2.   Enable the option **Implicit inclusion of source files for finding templates**.

Command line syntax

```
--implicit-include
```

Description

Enable implicit inclusion of source files as a method of finding definitions of template entities to be instantiated.

Related information

C++ compiler option **--instantiate** (Instantiation mode)

Section 2.4, *Template Instantiation*

C++ compiler option: --implicit-noexcept

Menu entry

-

Command line syntax

`--implicit-noexcept`

Description

With this option destructors and deallocation functions (i.e., operator `delete` and operator `delete[]`) have implicit `noexcept` specifications if no explicit exception specification is provided.

Related information

-

C++ compiler option: --incl-suffixes

Menu entry

-

Command line syntax

--incl-suffixes=`suffixes`

Default: no extension and `.stdh`.

Description

Specifies the list of suffixes to be used when searching for an include file whose name was specified without a suffix. If a null suffix is to be allowed, it must be included in the suffix list. *suffixes* is a colon-separated list of suffixes (e.g., "`::stdh`").

Example

To allow only the suffixes `.h` and `.stdh` as include file extensions, enter:

```
cptc --incl-suffixes=h:stdh test.cc
```

Related information

C++ compiler option **--include-file** (Include file at the start of a compilation)

Section 5.2, *How the C++ Compiler Searches Include Files*

C++ compiler option: --include-directory (-I)

Menu entry

1.   Select **C/C++ Compiler » Include Paths**.

     The Include paths box shows the directories that are added to the search path for include files.

2.   To define a new directory for the search path, click on the **Add** button in the **Include paths** box.

3.   Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

`--include-directory=`*path*`,...`

`-I`*path*`,...`

Description

Add *path* to the list of directories searched for `#include` files whose names do not have an absolute pathname. You can specify multiple directories separated by commas.

Example

To add the directory `/proj/include` to the include file search path, enter:

```
cptc --include-directory=/proj/include test.cc
```

Related information

C++ compiler option **--include-file** (Include file at the start of a compilation)

C++ compiler option **--sys-include** (Add directory to system include file search path)

Section 5.2, *How the C++ Compiler Searches Include Files*

C++ compiler option: --include-file (-H)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

    The Pre-include files box shows the files that are currently included before the compilation starts.

2. To define a new file, click on the **Add** button in the **Pre-include files** box.

3. Type the full path and file name or select a file.

Use the **Edit** and **Delete** button to change a file name or to remove a file from the list.

Command line syntax

```
--include-file=file
```

```
-Hfile
```

Description

Include the source code of the indicated file at the beginning of the compilation. This is the same as specifying `#include "file"` at the beginning of *each* of your C++ sources.

All files included with **--include-file** are processed after any of the files included with **--include-macros-file**.

The filename is searched for in the directories on the include search list.

Example

```
cptc --include-file=extra.h test1.cc test2.cc
```

The file `extra.h` is included at the beginning of both `test1.cc` and `test2.cc`.

Related information

C++ compiler option **--include-directory** (Add directory to include file search path)

Section 5.2, *How the C++ Compiler Searches Include Files*

C++ compiler option: --include-macros-file

Menu entry

-

Command line syntax

`--include-macros-file=`*file*

Description

Include the macros of the indicated file at the beginning of the compilation. Only the preprocessing directives from the file are evaluated. All of the actual code is discarded. The effect of this option is that any macro definitions from the specified file will be in effect when the primary source file is compiled. All of the macro-only files are processed before any of the normal includes (**--include-file**). Within each group, the files are processed in the order in which they were specified.

Related information

C++ compiler option **--include-file** (Include file at the start of a compilation)

Section 5.2, *How the C++ Compiler Searches Include Files*

C++ compiler option: --indirect

Menu entry

1. Select **C/C++ Compiler » Code Generation**.

2. Enable the option **Call functions indirectly**.

Command line syntax

--indirect

Description

With this option you tell the C++ compiler to generate code for indirect function calling.

In the generated C source, functions get the function qualifier `__indirect`.

Example

With the following command the C++ compiler generates far calls for all functions:

```
cptc --indirect test.cc
```

Related information

C compiler option **--indirect** (Call functions indirectly)

C++ compiler option: --init-priority

Menu entry

-

Command line syntax

`--init-priority=`*number*

Default: 0

Description

Normally, the C++ compiler assigns no priority to the global initialization functions and the exact order is determined by the linker. This option sets the default priority for global initialization functions. Default value is "0". You can also set the default priority with the `#pragma init_priority`.

Values from 1 to 100 are for internal use only and should not be used. Values 101 to 65535 are available for user code. A lower number means a higher priority.

Example

`cptc --init-priority=101 test.cc`

Related information

-

C++ compiler option: --inline-statement-limit

Menu entry

-

Command line syntax

`--inline-statement-limit=`*number*

Default: 100

Description

With this option you can set the maximum number of statements that a routine can have and still be inlinable. The setting is somewhat arbitrary, but a limit is needed to prevent iterative inlining from exhausting memory. A value of zero allows an inlined routine to be any number of statements.

Related information

-

C++ compiler option: --instantiate (-t)

Menu entry

1.   Select **C/C++ Compiler » Miscellaneous**.

2.   Select an instantiation mode in the **Instantiation mode of external template entities** box.

Command line syntax

--instantiate=*mode*

-t*mode*

You can specify the following modes:

> **used**
>
> **all**
>
> **local**

Default: **--instantiate=used**

Description

Control instantiation of external template entities. External template entities are external (that is, non-inline and non-static) template functions and template static data members. The instantiation mode determines the template entities for which code should be generated based on the template definition. Normally, when a file is compiled, template entities are instantiated wherever they are used (the linker will discard duplicate definitions). The overall instantiation mode can, however, be changed with this option. You can specify the following modes:

used	Instantiate those template entities that were used in the compilation. This will include all static data members for which there are template definitions. This is the default.
all	Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members will be instantiated whether or not they were used. Non-member template functions will be instantiated even if the only reference was a declaration.
local	Similar to **--instantiate=used** except that the functions are given internal linkage. This is intended to provide a very simple mechanism for those getting started with templates. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly (barring problems due to multiple copies of local static variables). However, one may end up with many copies of the instantiated functions, so this is not suitable for production use.

You cannot use **--instantiate=local** in conjunction with automatic template instantiation.

Related information

Section 2.4, *Template Instantiation*

C++ compiler option: --integer-enumeration

Menu entry

1.   Select **C/C++ Compiler » Language**.

2.   Enable the option **Always use 32-bit integers for enumeration**.

Command line syntax

```
--integer-enumeration
```

Description

Normally the C++ compiler treats enumerated types as the smallest data type possible (`char` or `short` instead of `int`). This reduces code size. With this option the C++ compiler always treats enum-types as `int` as defined in the ISO C99 standard.

Related information

Section 1.1, *Data Types*

C++ compiler option: --io-streams

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Support for C++ I/O streams**.

Command line syntax

`--io-streams`

Description

As I/O streams require substantial resources they are disabled by default. Use this option to enable I/O streams support in the C++ library. So, when you use standard I/O, e.g. `cout`, in your C++ source, you need to specify this option.

This option also enables exception handling.

Related information

-

C++ compiler option: --lambdas

Menu entry

-

Command line syntax

`--lambdas`

`--no-lambdas`

Default: **`--no-lambdas`**

Default in C++11 mode: **`--lambdas`**

Description

Enable or disable support for C++11 lambdas.

This feature is implicitly enabled in C++11 mode.

Related information

C++ compiler option **--c++11** (C++11 language extensions)

C++ compiler option: --late-tiebreaker

Menu entry

-

Command line syntax

`--late-tiebreaker`

Default: early tiebreaker processing.

Description

Select the way that tie-breakers (e.g., cv-qualifier differences) apply in overload resolution. In "early" tie-breaker processing, the tie-breakers are considered at the same time as other measures of the goodness of the match of an argument value and the corresponding parameter type (this is the standard approach).

In "late" tie-breaker processing, tie-breakers are ignored during the initial comparison, and considered only if two functions are otherwise equally good on all arguments; the tie-breakers can then be used to choose one function over another.

Related information

-

C++ compiler option: --list-file (-L)

Menu entry

-

Command line syntax

--list-file=_file_

-L_file_

Default: -1

Description

Generate raw listing information in the _file_. This information is likely to be used to generate a formatted listing. The raw listing file contains raw source lines, information on transitions into and out of include files, and diagnostics generated by the C++ compiler.

Each line of the listing file begins with a key character that identifies the type of line, as follows:

N   A normal line of source; the rest of the line is the text of the line.

X   The expanded form of a normal line of source; the rest of the line is the text of the line. This line appears following the N line, and only if the line contains non-trivial modifications (comments are considered trivial modifications; macro expansions, line splices, and trigraphs are considered non-trivial modifications). Comments are replaced by a single space in the expanded-form line.

S   A line of source skipped by an `#if` or the like; the rest of the line is text. Note that the `#else`, `#elif`, or `#endif` that ends a skip is marked with an N.

L   An indication of a change in source position. The line has a format similar to the # line-identifying directive output by the C preprocessor, that is to say

```
L   line_number "file-name" [key]
```

where _key_ is, **1** for entry into an include file, or **2** for exit from an include file, and omitted otherwise.

The first line in the raw listing file is always an L line identifying the primary input file. L lines are also output for #line directives (key is omitted). L lines indicate the source position of the following source line in the raw listing file.

R, **W**, **E**, or **C**   An indication of a diagnostic (**R** for remark, **W** for warning, **E** for error, and **C** for catastrophic error). The line has the form:

```
S  "file-name" line_number column-number message-text
```

where *S* is R, W, E, or C, as explained above. Errors at the end of file indicate the last line of the primary source file and a column number of zero. Command line errors are catastrophes with an empty file name ("") and a line and column number of zero. Internal errors are catastrophes with position information as usual, and message-text beginning with (internal error). When a diagnostic displays a list (e.g., all the contending routines when there is ambiguity on an overloaded call), the initial diagnostic line is followed by one or more lines with the same overall format (code letter, file name, line number, column number, and message text), but in which the code letter is the lowercase version of the code letter in the initial line. The source position in such lines is the same as that in the corresponding initial line.

Example

To write raw listing information to the file `test.lst`, enter:

```
cptc --list-file=test.lst test.cc
```

Related information

-

C++ compiler option: --long-lifetime-temps

Menu entry

-

Command line syntax

`--long-lifetime-temps`

Description

Select the lifetime for temporaries: short means to end of full expression; long means to the earliest of end of scope, end of switch clause, or the next label. Short is the default.

Related information

-

C++ compiler option: --long-long

Menu entry

-

Command line syntax

`--long-long`

Description

Permit the use of `long long` in strict mode in dialects in which it is non-standard.

Related information

-

C++ compiler option: --lossy-conversion-warning

Menu entry

-

Command line syntax

`--lossy-conversion-warning`

Description

With this option a warning is issued whenever a conversion occurs from one arithmetic type to a smaller one or from a floating type to an integral type.

Related information

-

C++ compiler option: --make-target

Menu entry

-

Command line syntax

`--make-target=`*name*

Description

With this option you can overrule the default target name in the make dependencies generated by the options **--preprocess=+make** (**-Em**) and **--dep-file**. The default target name is the basename of the input file, with extension `.o`.

Example

```
cptc --preprocess=+make --make-target=mytarget.o test.cc
```

The compiler generates dependency lines with the default target name `mytarget.o` instead of `test.o`.

Related information

C++ compiler option **--preprocess=+make** (Generate dependencies for make)

C++ compiler option **--dep-file** (Generate dependencies in a file)

C++ compiler option: --max-constexpr-call-count

Menu entry

-

Command line syntax

`--max-constexpr-call-count=``value`

Default: 50000

Description

With this option you can set the maximum number of `constexpr` function and constructor calls allowed in the expansion of one top-level call.

Related information

-

C++ compiler option: --max-constexpr-call-depth

Menu entry

-

Command line syntax

`--max-constexpr-call-depth=`*value*

Default: 1000

Description

With this option you can set the maximum nesting depth of `constexpr` function and constructor calls.

Related information

-

C++ compiler option: --mmu-present / --mmu-on

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor Selection** list, select a processor that has an MMU or select a user defined processor.

3. (Optional) Select **C/C++ Compiler » Miscellaneous**.

4. (Optional) Add the option **--mmu-present** to the **Additional options** field.

5. Select **C/C++ Compiler » Code Generation**.

6. Enable the option **Use the MMU if present**.

> For predefined TriCore processors, the option **--mmu-present** is set automatically, so you only need to enable the option **Use the MMU if present**. For user defined processors you need to enable them both.

Command line syntax

```
--mmu-present
--mmu-on
```

Description

If the processor you are using has an MMU which is activated, you can tell the C++ compiler to use the MMU. The C++ compiler then will align data naturally. To instruct the C++ compiler to use the MMU, you must set both the option **--mmu-present** and the option **--mmu-on**.

With the option **--mmu-present** you tell the C++ compiler that an MMU is physically present. With the option **--mmu-on** you tell the C++ compiler the MMU is activated. In this case the C++ compiler needs to align data naturally. Both options are necessary.

If you select a valid target processor, the option **--mmu-present** is set automatically, based on the chosen target processor. If you are using a target processor with MMU that is not predefined, you need to set this option manually.

Example

To instruct the C++ compiler to use the activated MMU:

```
cptc --mmu-present --mmu-on test.cc
```

Related information

Control program option **--cpu** (Select processor)

C++ compiler option **--no-fpu** (Disable hardware floating-point instructions)

C++ compiler option: --multibyte-chars

Menu entry

-

Command line syntax

`--multibyte-chars`

Default: multibyte character sequences are not allowed.

Description

Enable processing for multibyte character sequences in comments, string literals, and character constants. Multibyte encodings are used for character sets like the Japanese SJIS.

Related information

-

C++ compiler option: --namespaces

Menu entry

-

Command line syntax

`--namespaces`

`--no-namespaces`

Default: namespaces are supported.

Description

When you used option **--embedded-c++** namespaces are disabled. With option **--namespaces** you can enable support for namespaces in this case.

The macro __NAMESPACES is defined when namespace support is enabled.

Related information

C++ compiler option **--embedded-c++** (Embedded C++ compliancy tests)

C++ compiler option **--using-std** (Implicit use of the std namespace)

Section 2.3, *Namespace Support*

C++ compiler option: --no-arg-dep-lookup

Menu entry

-

Command line syntax

`--no-arg-dep-lookup`

Default: argument dependent lookup of unqualified function names is performed.

Description

With this option you disable argument dependent lookup of unqualified function names.

Related information

-

C++ compiler option: --no-array-new-and-delete

Menu entry

-

Command line syntax

`--no-array-new-and-delete`

Default: array new and delete are supported.

Description

Disable support for array new and delete.

The macro `__ARRAY_OPERATORS` is defined when array new and delete is enabled.

Related information

-

C++ compiler option: --no-auto-storage

Menu entry

-

Command line syntax

`--no-auto-storage`

Default: `auto` is a storage class specifier.

Description

Disable the traditional meaning of `auto` keyword as a storage class specifier.

Related information

C++ compiler option **--auto-type** (`auto` can be used as type specifier)

C++ compiler option: --no-bool

Menu entry

-

Command line syntax

`--no-bool`

Default: `bool` is recognized as a keyword.

Description

Disable recognition of the `bool` keyword.

The macro `_BOOL` is defined when `bool` is recognized as a keyword.

Related information

-

C++ compiler option: --no-class-name-injection

Menu entry

-

Command line syntax

`--no-class-name-injection`

Default: the name of a class is injected into the scope of the class (as required by the standard).

Description

Do not inject the name of a class into the scope of the class (as was true in earlier versions of the C++ language).

Related information

-

C++ compiler option: --no-const-string-literals

Menu entry

-

Command line syntax

`--no-const-string-literals`

Default: C++ string literals and wide string literals are `const` (as required by the standard).

Description

With this option C++ string literals and wide string literals are non-const (as was true in earlier versions of the C++ language).

Related information

-

C++ compiler option: --no-dep-name

Menu entry

-

Command line syntax

`--no-dep-name`

Default: dependent name processing is enabled.

Description

Disable dependent name processing; i.e., the special lookup of names used in templates as required by the C++ standard. This option implies the use of **--no-parse-templates**.

Related information

C++ compiler option **--no-parse-templates** (Disable parsing of nonclass templates)

C++ compiler option: --no-distinct-template-signatures

Menu entry

-

Command line syntax

`--no-distinct-template-signatures`

Description

Control whether the signatures for template functions can match those for non-template functions when the functions appear in different compilation units. By default a normal function cannot be used to satisfy the need for a template instance; e.g., a function "`void f(int)`" could not be used to satisfy the need for an instantiation of a template "`void f(T)`" with `T` set to `int`.

--no-distinct-template-signatures provides the older language behavior, under which a non-template function can match a template function. Also controls whether function templates may have template parameters that are not used in the function signature of the function template.

Related information

-

C++ compiler option: --no-double (-F)

Menu entry

1. Select **C/C++ Compiler » Floating-Point**.

2. Select **Custom** from the **Floating-point model** list.

3. Enable the option **Treat 'double' as 'float'**.

Command line syntax

```
--no-double
```

```
-F
```

Description

With this option you tell the C++ compiler to treat variables of the type double as float. Because the float type takes less space, execution speed increases and code size decreases, both at the cost of less precision.

Example

```
cptc --no-double test.cc
```

The file test.cc is compiled where variables of the type double are treated as float.

Related information

-

C++ compiler option: --no-enum-overloading

Menu entry

-

Command line syntax

`--no-enum-overloading`

Description

Disable support for using operator functions to overload built-in operations on enum-typed operands.

Related information

-

C++ compiler option: --no-explicit

Menu entry

-

Command line syntax

`--no-explicit`

Default: the `explicit` specifier is allowed.

Description

Disable support for the `explicit` specifier on constructor declarations.

Related information

-

C++ compiler option: --no-extern-inline

Menu entry

-

Command line syntax

`--no-extern-inline`

Default: `inline` functions are allowed to have external linkage.

Description

Disable support for `inline` functions with external linkage in C++. When `inline` functions are allowed to have external linkage (as required by the standard), then `extern` and `inline` are compatible specifiers on a non-member function declaration; the default linkage when `inline` appears alone is external (that is, `inline` means `extern inline` on non-member functions); and an `inline` member function takes on the linkage of its class (which is usually external). However, when `inline` functions have only internal linkage (using **--no-extern-inline**), then `extern` and `inline` are incompatible; the default linkage when `inline` appears alone is internal (that is, `inline` means `static inline` on non-member functions); and `inline` member functions have internal linkage no matter what the linkage of their class.

Related information

Section 2.6, *Extern Inline Functions*

C++ compiler option: --no-for-init-diff-warning

Menu entry

-

Command line syntax

`--no-for-init-diff-warning`

Description

Disable a warning that is issued when programs compiled without the **--old-for-init** option would have had different behavior under the old rules.

Related information

C++ compiler option **--old-for-init** (Use old for scoping rules)

C++ compiler option: --no-fpu

Menu entry

1. Select **C/C++ Compiler » Code Generation**.

2. Disable the option **Use the FPU if present**.

Command line syntax

--no-fpu

Description

By default, the C++ compiler assumes an FPU is present and the macro __FPU__ is defined in the C++ source file. With this option the C++ compiler does not generate single precision floating-point instructions in the assembly file. When you select this option, the macro __FPU__ is not defined in the C++ source file.

If you select a valid target processor, this option is automatically set, based on the chosen target processor.

Example

To disable the use of floating-point unit (FPU) instructions in the assembly code, enter:

```
cptc --no-fpu test.cc
```

Related information

Control program option **--cpu** (Select processor)

C++ compiler option **--core** (Select TriCore architecture)

C++ compiler option: --no-implicit-typename

Menu entry

-

Command line syntax

`--no-implicit-typename`

Default: implicit typename determination is enabled.

Description

Disable implicit determination, from context, whether a template parameter dependent name is a type or nontype.

Related information

C++ compiler option **--no-typename** (Disable the typename keyword)

C++ compiler option: --no-inlining

Menu entry

1.   Select **C/C++ Compiler » Miscellaneous**.

2.   Disable the option **Minimal inlining of function calls (C++)**.

Command line syntax

`--no-inlining`

Description

Disable minimal inlining of function calls.

Related information

-

C++ compiler option: --nonconst-ref-anachronism

Menu entry

-

Command line syntax

`--nonconst-ref-anachronism`

`--no-nonconst-ref-anachronism`

Default: `--no-nonconst-ref-anachronism`

Description

Enable or disable the anachronism of allowing a reference to nonconst to bind to a class rvalue of the right type. This anachronism is also enabled by the **--anachronisms** option.

Related information

C++ compiler option **--anachronisms** (Enable C++ anachronisms)

Section 2.2.3, *Anachronisms Accepted*

C++ compiler option: --nonstd-anonymous-unions

Menu entry

-

Command line syntax

`--nonstd-anonymous-unions`

Description

Enable extensions to the C++ anonymous union feature (including of anonymous "structs" or "classes").

Related information

-

C++ compiler option: --nonstd-default-arg-deduction

Menu entry

-

Command line syntax

`--nonstd-default-arg-deduction`

Description

Controls whether default arguments are retained as part of deduced function types. The C++ standard requires that default arguments not be part of deduced function types. This option changes that behavior.

Related information

-

C++ compiler option: --nonstd-instantiation-lookup

Menu entry

-

Command line syntax

`--nonstd-instantiation-lookup`

Description

Controls whether the lookup of names during template instantiation should, instead of the normal lookup rules, use rules that were part of the C++98 working paper for some time during the development of the standard. In this mode, names are looked up in both the namespace of the template definition and in the namespace in which a template entity was first referenced in a way that would require an instantiation.

Related information

-

C++ compiler option: --nonstd-qualifier-deduction

Menu entry

-

Command line syntax

`--nonstd-qualifier-deduction`

Description

Controls whether non-standard template argument deduction should be performed in the qualifier portion of a qualified name. With this feature enabled, a template argument for the template parameter `T` can be deduced in contexts like `A<T>::B` or `T::B`. The standard deduction mechanism treats these as non-deduced contexts that use the values of template parameters that were either explicitly specified or deduced elsewhere.

Related information

-

C++ compiler option: --nonstd-using-decl

Menu entry

-

Command line syntax

`--nonstd-using-decl`

Default: non-standard using declarations are not allowed.

Description

Allow a non-member using declaration that specifies an unqualified name.

Related information

-

C++ compiler option: --no-parse-templates

Menu entry

-

Command line syntax

`--no-parse-templates`

Default: parsing of nonclass templates is enabled.

Description

Disable the parsing of nonclass templates in their generic form (i.e., even if they are not really instantiated). It is done by default if dependent name processing is enabled.

Related information

C++ compiler option **--no-dep-name** (Disable dependent name processing)

C++ compiler option: --no-pch-messages

Menu entry

-

Command line syntax

`--no-pch-messages`

Default: a message is displayed indicating that a precompiled header file was created or used in the current compilation. For example,

`"test.cc": creating precompiled header file "test.pch"`

Description

Disable the display of a message indicating that a precompiled header file was created or used in the current compilation.

Related information

C++ compiler option **--pch** (Automatic PCH mode)

C++ compiler option **--use-pch** (Use precompiled header file)

C++ compiler option **--create-pch** (Create precompiled header file)

Section 2.9, *Precompiled Headers*

C++ compiler option: --no-preprocessing-only

Menu entry

Eclipse always does a full compilation.

Command line syntax

`--no-preprocessing-only`

Description

You can use this option in conjunction with the options that normally cause the C++ compiler to do preprocessing only (e.g., **--preprocess**, etc.) to specify that a full compilation should be done (not just preprocessing). When used with the implicit inclusion option, this makes it possible to generate a preprocessed output file that includes any implicitly included files.

Example

`cptc --preprocess --implicit-include --no-preprocessing-only test.cc`

Related information

C++ compiler option **--preprocess** (Preprocessing only)

C++ compiler option **--implicit-include** (Implicit source file inclusion)

C++ compiler option: --no-stdarg-builtin

Menu entry

-

Command line syntax

`--no-stdarg-builtin`

Description

Disable special treatment of the stdarg.h header. When enabled, the stdarg.h header is treated as a built-in, and references to its macros ("va_start" et al) are passed through as such in generated C code.

Related information

-

C++ compiler option: --no-stdinc / --no-stdstlinc

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--no-stdinc** or **--no-stdstlinc** to the **Additional options** field.

Command line syntax

```
--no-stdinc
```

```
--no-stdstlinc
```

Description

With option **--no-stdinc** you tell the C++ compiler not to look in the default include directory relative to the installation directory, when searching for standard include files.

With option **--no-stdstlinc** you tell the C++ compiler not to look in the default include.stl directory relative to the installation directory, when searching for standard STL include files.

This way the C++ compiler only searches in the include file search paths you specified.

Related information

Section 5.2, *How the C++ Compiler Searches Include Files*

C++ compiler option: --no-typename

Menu entry

-

Command line syntax

`--no-typename`

Default: `typename` is recognized as a keyword.

Description

Disable recognition of the `typename` keyword.

Related information

C++ compiler option **--no-implicit-typename** (Disable implicit typename determination)

C++ compiler option: --no-use-before-set-warnings (-j)

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

2. Enable the option **Suppress C++ compiler "used before set" warnings**.

Command line syntax

```
--no-use-before-set-warnings
```

```
-j
```

Description

Suppress warnings on local automatic variables that are used before their values are set.

Related information

C++ compiler option **--no-warnings** (Suppress all warnings)

C++ compiler option: --no-warnings (-w)

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

2. Enable the option **Suppress all warnings**.

Command line syntax

```
--no-warnings
```

```
-w
```

Description

With this option you suppress all warning messages. Error messages are still issued.

Related information

C++ compiler option **--warnings-as-errors** (Treat warnings as errors)

C++ compiler option: --nullptr

Menu entry

-

Command line syntax

`--nullptr`

`--no-nullptr`

Default: **`--no-nullptr`**

Default in C++11 mode: **`--nullptr`**

Description

Enable or disable support for the C++11 `nullptr` keyword.

This feature is implicitly enabled in C++11 mode.

Related information

C++ compiler option **--c++11** (C++11 language extensions)

C++ compiler option: --old-for-init

Menu entry

-

Command line syntax

`--old-for-init`

Description

Control the scope of a declaration in a `for-init-statement`. The old (cfront-compatible) scoping rules mean the declaration is in the scope to which the `for` statement itself belongs; the default (standard-conforming) rules in effect wrap the entire `for` statement in its own implicitly generated scope.

Related information

C++ compiler option **--no-for-init-diff-warning** (Disable warning for old for-scoping)

C++ compiler option: --old-line-commands

Menu entry

-

Command line syntax

`--old-line-commands`

Description

When generating source output, put out #line directives in the form # *nnn* instead of #line *nnn*.

Example

To do preprocessing only, without comments and with old style line control information, enter:

```
cptc --preprocess --old-line-commands test.cc
```

Related information

C++ compiler option **--preprocess** (Preprocessing only)

C++ compiler option: --old-specializations

Menu entry

-

Command line syntax

`--old-specializations`

Description

Enable acceptance of old-style template specializations (that is, specializations that do not use the `template<>` syntax).

Related information

-

C++ compiler option: --option-file (-f)

Menu entry

-

Command line syntax

--option-file=*file*

-f *file*

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the C++ compiler.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** multiple times.

Format of an option file

• Multiple arguments on one line in the option file are allowed.

• To include whitespace in an argument, surround the argument with single or double quotes.

• If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

```
"This has a single quote ' embedded"

'This has a double quote " embedded'

'This has a double quote " and a single quote '"' embedded"
```

• When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \
line"

        -> "This is a continuation line"
```

• It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
--embedded-c++
--define=DEMO=1
test.cc
```

Specify the option file to the C++ compiler:

```
cptc --option-file=myoptions
```

This is equivalent to the following command line:

```
cptc --embedded-c++ --define=DEMO=1 test.cc
```

Related information

-

C++ compiler option: --output (-o)

Menu entry

Eclipse names the output file always after the C++ source file.

Command line syntax

--output-file=*file*

-o *file*

Default: module name with `.ic` suffix.

Description

With this option you can specify another filename for the output file of the C++ compiler. Without this option the basename of the C++ source file is used with extension `.ic`.

You can also use this option in combination with the option **--preprocess (-E)** to redirect the preprocessing output to a file.

Example

To create the file `output.ic` instead of `test.ic`, enter:

```
cptc --output=output.ic test.cc
```

To use the file `my.pre` as the preprocessing output file, enter:

```
cptc --preprocess --output=my.pre test.cc
```

Related information

C++ compiler option **--preprocess** (Preprocessing)

C++ compiler option: --pch

Menu entry

1. Select **C/C++ Compiler » Precompiled C++ Headers**.

2. Enable the option **Automatically use/create precompiled header file**.

Command line syntax

`--pch`

Description

Automatically use and/or create a precompiled header file. If **--use-pch** or **--create-pch** (manual PCH mode) appears on the command line following this option, its effect is erased.

Related information

C++ compiler option **--use-pch** (Use precompiled header file)

C++ compiler option **--create-pch** (Create precompiled header file)

Section 2.9, *Precompiled Headers*

C++ compiler option: --pch-dir

Menu entry

1. Select **C/C++ Compiler » Precompiled C++ Headers**.

2. Enter a path in the **Precompiled header file directory**.

Command line syntax

`--pch-dir=`*directory-name*

Description

Specify the directory in which to search for and/or create a precompiled header file. This option may be used with automatic PCH mode (**--pch**) or manual PCH mode (**--create-pch** or **--use-pch**).

Example

To use the directory `/usr/include/pch` to automatically create precompiled header files, enter:

```
cptc --pch-dir=/usr/include/pch --pch test.cc
```

Related information

C++ compiler option **--pch** (Automatic PCH mode)

C++ compiler option **--use-pch** (Use precompiled header file)

C++ compiler option **--create-pch** (Create precompiled header file)

Section 2.9, *Precompiled Headers*

C++ compiler option: --pch-verbose

Menu entry

-

Command line syntax

`--pch-verbose`

Description

In automatic PCH mode, for each precompiled header file that cannot be used for the current compilation, a message is displayed giving the reason that the file cannot be used.

Example

```
cptc --pch --pch-verbose test.cc
```

Related information

C++ compiler option **--pch** (Automatic PCH mode)

Section 2.9, *Precompiled Headers*

C++ compiler option: --pending-instantiations

Menu entry

-

Command line syntax

`--pending-instantiations=`n

where n is the maximum number of instantiations of a single template.

Default: 64

Description

Specifies the maximum number of instantiations of a given template that may be in process of being instantiated at a given time. This is used to detect runaway recursive instantiations. If n is zero, there is no limit.

Example

To specify a maximum of 32 pending instantiations, enter:

```
cptc --pending-instantiations=32 test.cc
```

Related information

Section 2.4, *Template Instantiation*

C++ compiler option: --preprocess (-E)

Menu entry

1.   Select **C/C++ Compiler » Preprocessing**.

2.   Enable the option **Store preprocessor output in <file>.pre**.

3.   (Optional) Enable the option **Keep comments in preprocessor output**.

4.   (Optional) Enable the option **Keep #line info in preprocessor output**.

Command line syntax

--preprocess[**=***flags*]

-E[*flags*]

You can set the following flags:

+/-comments	**c/C**	keep comments
+/-includes	**i/I**	generate a list of included source files
+/-make	**m/M**	generate dependencies for make
+/-noline	**p/P**	strip #line source position information

Default: **-ECIMP**

Description

With this option you tell the C++ compiler to preprocess the C++ source. Under Eclipse the C++ compiler sends the preprocessed output to the file *name*.pre (where *name* is the name of the C++ source file to compile). Eclipse also compiles the C++ source.

On the command line, the C++ compiler sends the preprocessed file to stdout. To capture the information in a file, specify an output file with the option **--output**.

With **--preprocess=+comments** you tell the preprocessor to keep the comments from the C++ source file in the preprocessed output.

With **--preprocess=+includes** the C++ compiler will generate a list of all included source files. The preprocessor output is discarded.

With **--preprocess=+make** the C++ compiler will generate dependency lines that can be used in a Makefile. The preprocessor output is discarded. The default target name is the basename of the input file, with the extension .o. With the option **--make-target** you can specify a target name which overrules the default target name.

When implicit inclusion of templates is enabled, the output may indicate false (but safe) dependencies unless **--no-preprocessing-only** is also used.

With **--preprocess=+noline** you tell the preprocessor to strip the #line source position information (lines starting with #line). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is easier to read.

Example

```
cptc --preprocess=+comments,-make,-noline test.cc --output=test.pre
```

The C++ compiler preprocesses the file test.cc and sends the output to the file test.pre. Comments are included but no dependencies are generated and the line source position information is not stripped from the output file.

Related information

C++ compiler option **--no-preprocessing-only** (Force full compilation)

C++ compiler option **--dep-file** (Generate dependencies in a file)

C++ compiler option **--make-target** (Specify target name for **-Em** output)

C++ compiler option: --preserve-lvalues-with-same-type-casts

Menu entry

-

Command line syntax

`--preserve-lvalues-with-same-type-casts`

Description

With this option a cast of an lvalue to its own type results in an lvalue (rather than an rvalue, as required by the C++ standard).

Related information

-

C++ compiler option: --remarks (-r)

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

2. Enable the option **Issue remarks on C++ code**.

Command line syntax

```
--remarks
```

```
-r
```

Description

Issue remarks, which are diagnostic messages even milder than warnings.

Related information

Section 5.3, *C++ Compiler Error Messages*

C++ compiler option: --remove-unneeded-entities

Menu entry

-

Command line syntax

`--remove-unneeded-entities`

Description

Enable an optimization to remove types, variables, routines, and related constructs that are not really needed. Something may be referenced but unneeded if it is referenced only by something that is itself unneeded; certain entities, such as global variables and routines defined in the translation unit, are always considered to be needed.

Related information

-

C++ compiler option: --rtti

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Support for C++ RTTI (run-time type information)**.

Command line syntax

`--rtti`

Default: RTTI (run-time type information) features are disabled.

Description

Enable support for RTTI (run-time type information) features: `dynamic_cast`, `typeid`.

The macro `__RTTI` is defined when RTTI support is enabled.

Related information

-

C++ compiler option: --rvalue-ctor-is-not-copy-ctor

Menu entry

-

Command line syntax

`--rvalue-ctor-is-not-copy-ctor`

Description

Do not treat an rvalue (or "move") constructor as a copy constructor. In the default situation, where rvalue constructors are treated as copy constructors, a user-declared rvalue constructor will inhibit the implicit generation of a traditional copy constructor.

Related information

-

C++ compiler option: --rvalue-refs

Menu entry

-

Command line syntax

`--rvalue-refs`

`--no-rvalue-refs`

Default: **`--no-rvalue-refs`**

Default in C++11 mode: **`--rvalue-refs`**

Description

Enable or disable support for rvalue references.

This feature is implicitly enabled in C++11 mode.

Related information

C++ compiler option **--c++11** (C++11 language extensions)

C++ compiler option: --schar (-s)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Disable the option **Treat 'char' variables as unsigned**.

Command line syntax

`--schar`

`-s`

Description

With this option `char` is the same as `signed char`. This is the default.

When plain `char` is signed, the macro `__SIGNED_CHARS__` is defined.

Related information

C++ compiler option **--uchar** (Plain char is unsigned)

Section 1.1, *Data Types*

C++ compiler option: --signed-bitfields

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat 'int' bit-fields as signed**.

Command line syntax

`--signed-bitfields`

Description

For bit-fields it depends on the implementation whether a plain `int` is treated as `signed int` or `unsigned int`. By default an `int` bit-field is treated as `unsigned int`. This offers the best performance. With this option you tell the C++ compiler to treat `int` bit-fields as `signed int`. In this case, you can still add the keyword `unsigned` to treat a particular `int` bit-field as `unsigned`.

Related information

C compiler option **--signed-bitfields**

C++ compiler option **--unsigned-bitfields**

Section 1.1, *Data Types*

C++ compiler option: --special-subscript-cost

Menu entry

-

Command line syntax

`--special-subscript-cost`

Description

Enable a special nonstandard weighting of the conversion to the integral operand of the `[]` operator in overload resolution.

This is a compatibility feature that may be useful with some existing code. With this feature enabled, the following code compiles without error:

```
struct A {
    A();
    operator int *();
    int operator[](unsigned);
};
void main() {
    A a;
    a[0];   // Ambiguous, but allowed with this option
            // operator[] is chosen
}
```

Related information

-

C++ compiler option: --strict (-A)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Disable the option **Allow non-ANSI/ISO C++ features**.

Command line syntax

`--strict`

`-A`

Default: non-ANSI/ISO C++ features are enabled.

Description

Enable strict ANSI/ISO mode, which provides diagnostic messages when non-standard features are used, and disables features that conflict with ANSI/ISO C or C++. All ANSI/ISO violations are issued as errors.

Example

To enable strict ANSI mode, with error diagnostic messages, enter:

```
cptc --strict test.cc
```

Related information

C++ compiler option **--strict-warnings** (Strict ANSI/ISO mode with warnings)

C++ compiler option: --strict-warnings (-a)

Menu entry

-

Command line syntax

`--strict-warnings`

`-a`

Default: non-ANSI/ISO C++ features are enabled.

Description

This option is similar to the option **--strict**, but all violations are issued as warnings instead of errors.

Example

To enable strict ANSI mode, with warning diagnostic messages, enter:

```
cptc --strict-warnings test.cc
```

Related information

C++ compiler option **--strict** (Strict ANSI/ISO mode with errors)

C++ compiler option: --suppress-vtbl

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Enable the option **Suppress definition of virtual function tables (C++)**.

Command line syntax

```
--suppress-vtbl
```

Description

Suppress definition of virtual function tables in cases where the heuristic used by the C++ compiler to decide on definition of virtual function tables provides no guidance. The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first non-inline non-pure virtual function of the class. For classes that contain no such function, the default behavior is to define the virtual function table (but to define it as a local static entity). The **--suppress-vtbl** option suppresses the definition of the virtual function tables for such classes, and the **--force-vtbl** option forces the definition of the virtual function table for such classes. **--force-vtbl** differs from the default behavior in that it does not force the definition to be local.

Related information

C++ compiler option **--force-vtbl** (Force definition of virtual function tables)

C++ compiler option: --sys-include

Menu entry

-

Command line syntax

--sys-include=*directory*,...

Description

Change the algorithm for searching system include files whose names do not have an absolute pathname to look in *directory*.

Example

To add the directory /proj/include to the system include file search path, enter:

```
cptc --sys-include=/proj/include test.cc
```

Related information

C++ compiler option **--include-directory** (Add directory to include file search path)

Section 5.2, *How the C++ Compiler Searches Include Files*

C++ compiler option: --template-typedefs-in-diagnostics

Menu entry

-

Command line syntax

`--template-typedefs-in-diagnostics`

Default: typedefs declared in template classes are not replaced with their underlying type.

Description

Enable the replacement of typedefs declared in template classes with their underlying type. Diagnostic messages are often more useful when such typedefs are replaced.

Related information

-

C++ compiler option: --timing

Menu entry

-

Command line syntax

`--timing`

Default: no timing information is generated.

Description

Generate compilation timing information. This option causes the C++ compiler to display the amount of CPU time and elapsed time used by each phase of the compilation and a total for the entire compilation.

Example

```
cptc --timing test.cc

processed 180 lines at 8102 lines/min
```

Related information

-

C++ compiler option: --trace-includes

Menu entry

-

Command line syntax

`--trace-includes`

Description

Output a list of the names of files #included to the error output file. The source file is compiled normally (i.e. it is not just preprocessed) unless another option that causes preprocessing only is specified.

Example

```
cptc --trace-includes test.cc

iostream.h
string.h
```

Related information

C++ compiler option **--preprocess** (Preprocessing only)

C++ compiler option: --type-traits-helpers

Menu entry

-

Command line syntax

`--type-traits-helpers`

`--no-type-traits-helpers`

Default: in C++ mode type traits helpers are enabled by default. In GNU C++ mode, type traits helpers are never enabled by default.

Description

Enable or disable type traits helpers (like `__is_union` and `__has_virtual_destructor`). Type traits helpers are meant to ease the implementation of ISO/IEC TR 19768.

The macro `__TYPE_TRAITS_ENABLED` is defined when type traits pseudo-functions are enabled.

Related information

-

C++ compiler option: --uchar (-u)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat 'char' variables as unsigned**.

Command line syntax

`--uchar`

`-u`

Description

By default `char` is the same as specifying `signed char`. With this option `char` is the same as `unsigned char`.

Related information

C++ compiler option **--schar** (Plain char is signed)

Section 1.1, *Data Types*

C++ compiler option: --uliterals

Menu entry

-

Command line syntax

`--uliterals`

Default: U-literals are not recognized.

Description

Enable recognition of U-literals (string literals of the forms `U"..."` and `u"..."`, and character literals of the forms `U'...'` and `u'...'`). In C++ mode, this also enables the C++11 `char16_t` and `char32_t` keywords.

Related information

-

C++ compiler option: --undefine (-U)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**

    The Defined symbols box shows the symbols that are currently defined.

2. To remove a defined symbol, select the symbol in the **Defined symbols** box and click on the **Delete** button.

Command line syntax

--undefine=`macro_name`

-U`macro_name`

Description

Remove any initial definition of *macro_name* as in #undef. **--undefine** options are processed after all **--define** options have been processed.

You cannot undefine a predefined macro as specified in Section 2.8, *Predefined Macros*, except for:

    __STDC__
    __cplusplus
    __SIGNED_CHARS__

Example

To undefine the predefined macro __cplusplus:

```
cptc --undefine=__cplusplus test.cc
```

Related information

C++ compiler option **--define** (Define preprocessor macro)

Section 2.8, *Predefined Macros*

C++ compiler option: --unrestricted-unions

Menu entry

-

Command line syntax

`--unrestricted-unions`

Description

Accept C++11-style "unrestricted unions". For example, non-static data members of unions can have nontrivial constructors and destructors if you use this option.

Related information

-

C++ compiler option: --unsigned-bitfields

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Disable the option **Treat 'int' bit-fields as signed**.

Command line syntax

```
--unsigned-bitfields
```

Description

For bit-fields it depends on the implementation whether a plain `int` is treated as `signed int` or `unsigned int`. By default (this option) an `int` bit-field is treated as `unsigned int`. This offers the best performance.

Related information

C++ compiler option **--signed-bitfields**

C compiler option **--signed-bitfields**

Section 1.1, *Data Types*

C++ compiler option: --use-pch

Menu entry

1. Select **C/C++ Compiler » Precompiled C++ Headers**.

2. Enter a filename in the **Use precompiled header file** field.

Command line syntax

--use-pch=*filename*

Description

Use a precompiled header file of the specified name as part of the current compilation. If **--pch** (automatic PCH mode) or **--create-pch** appears on the command line following this option, its effect is erased.

Example

To use the precompiled header file with the name `test.pch`, enter:

```
cptc --use-pch=test.pch test.cc
```

Related information

C++ compiler option **--pch** (Automatic PCH mode)

C++ compiler option **--create-pch** (Create precompiled header file)

Section 2.9, *Precompiled Headers*

C++ compiler option: --user-defined-literals

Menu entry

-

Command line syntax

`--user-defined-literals`

Description

Allow C++11-style "user-defined literals", i.e., literals with a suffix designating an `operator` function that is to be called to produce the value of the literal, e.g., `12.34_km`, which calls `operator ""_km(12.34)`.

Related information

-

C++ compiler option: --using-std

Menu entry

-

Command line syntax

`--using-std`

Default: implicit use of the `std` namespace is disabled.

Description

Enable implicit use of the `std` namespace when standard header files are included. Note that this does not do the equivalent of putting a "`using namespace std;`" in the program to allow old programs to be compiled with new header files; it has a special and localized meaning related to the TASKING versions of certain header files, and is unlikely to be of much use to end-users of the TASKING C++ compiler.

Related information

C++ compiler option **--namespaces** (Support for namespaces)

Section 2.3, *Namespace Support*

C++ compiler option: --variadic-macros

Menu entry

-

Command line syntax

`--variadic-macros`

`--no-variadic-macros`

Default: `--no-variadic-macros`

Default in C++11 mode: `--variadic-macros`

Description

Allow or do not allow macros with a variable number of arguments.

This feature is implicitly enabled in C++11 mode.

Related information

C++ compiler option **--extended-variadic-macros** (Allow extended variadic macros)

C++ compiler option **--c++11** (C++11 language extensions)

C++ compiler option: --variadic-templates

Menu entry

-

Command line syntax

`--variadic-templates`

Description

Allow "variadic templates", a C++11 feature that allows declaration of templates with a variable number of arguments.

Related information

-

C++ compiler option: --version (-V)

Menu entry

-

Command line syntax

--version

-V

Description

Display version information. The C++ compiler ignores all other options or input files.

Example

```
cptc --version
```

The C++ compiler does not compile any files but displays the following version information:

```
TASKING VX-toolset for TriCore: C++ compiler    vx.yrz Build nnn
Copyright 2002-year Altium BV                   Serial# 00000000
```

Related information

-

C++ compiler option: --vla

Menu entry

-

Command line syntax

`--vla`

Description

Enable support for variable length arrays. This allows the declaration and use of arrays of automatic storage duration with dimensions that are fixed at run-time.

C++ compiler option: --warnings-as-errors

Menu entry

1.   Select **Global Options**.

2.   Enable the option **Treat warnings as errors**.

Command line syntax

`--warnings-as-errors`[`=`*number,...*]

Description

If the C++ compiler encounters an error, it stops compiling. When you use this option without arguments, you tell the C++ compiler to treat all warnings as errors. This means that the exit status of the C++ compiler will be non-zero after one or more compiler warnings. As a consequence, the C++ compiler now also stops after encountering a warning.

You can limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

C++ compiler option **--no-warnings** (Suppress all warnings)

C++ compiler option: --wchar_t-keyword

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Allow the 'wchar_t' keyword (C++)**.

Command line syntax

`--wchar_t-keyword`

Default: `wchar_t` is not recognized as a keyword.

Description

Enable recognition of `wchar_t` as a keyword.

The macro `_WCHAR_T` is defined when `wchar_t` is recognized as a keyword.

Related information

-

C++ compiler option: --xref-file (-X)

Menu entry

-

Command line syntax

--xref-file=*file*

-X*file*

Description

Generate cross-reference information in a *file*. For each reference to an identifier in the source program, a line of the form

symbol_id name X file-name line-number column-number

is written, where *X* is

D	for definition;
d	for declaration (that is, a declaration that is not a definition);
M	for modification;
A	for address taken;
U	for used;
C	for changed (but actually meaning used and modified in a single operation, such as an increment);
R	for any other kind of reference, or
E	for an error in which the kind of reference is indeterminate.

symbol-id is a unique decimal number for the symbol. The fields of the above line are separated by tab characters.

Related information

-

10.4. Assembler Options

This section lists all assembler options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the assembler via the control program. Therefore, it uses the syntax of the control program to pass options and files to the assembler. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **Assembler » Miscellaneous**.

4. In the **Additional options** field, enter one or more command line options.

    *Because Eclipse uses the control program, Eclipse automatically precedes the option with **-Wa** to pass the option via the control program directly to the assembler.*

Note that the options you enter in the Assembler page are not only used for hand-coded assembly files, but also for the assembly files generated by the compiler.

Be aware that some command line options are not useful in Eclipse or just do not have any effect. For example, the option **-V** displays version header information and has no effect in Eclipse.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+***longflag*. To switch a flag off, use an uppercase letter or a **-***longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
astc -Ogs test.src
astc --optimize=+generics,+instr-size test.src
```

When you do not specify an option, a default value may become active.

Assembler option: --case-insensitive (-c)

Menu entry

1. Select **Assembler » Symbols**.

2. Enable the option **Case insensitive identifiers**.

Command line syntax

`--case-insensitive`

`-c`

Default: case sensitive

Description

With this option you tell the assembler not to distinguish between uppercase and lowercase characters. By default the assembler considers uppercase and lowercase characters as different characters.

Assembly source files that are generated by the compiler must always be assembled case sensitive. When you are writing your own assembly code, you may want to specify the case insensitive mode.

Example

When assembling case insensitive, the label `LabelName` is the same label as `labelname`.

```
astc --case-insensitive test.src
```

Related information

Assembler control **$CASE**

Assembler option: --check

Menu entry

-

Command line syntax

`--check`

Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application.

The assembler reports any warnings and/or errors.

This option is available on the command line only.

Related information

C compiler option **--check** (Check syntax)

Assembler option: --concatenate-sections

Menu entry

-

Command line syntax

`--concatenate-sections`

Description

By default the linker does not merge sections with the same name into one section. With this option the assembler uses the section attribute `concat`, instructing the linker to merge sections with the same name.

The advantage of section concatenation is faster locating, because there are less sections to locate.

The disadvantage of section concatenation is less efficient memory use, because of alignment gaps between (sequentially concatenated) sections.

Related information

-

Assembler option: --core

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor Selection** list, select a processor.

Command line syntax

--core=_core_

You can specify the following _core_ arguments:

tc1.3	TriCore 1.3 architecture
tc1.3.1	TriCore 1.3.1 architecture, TriCore 1.3.1 instructions are allowed
tc1.6	TriCore 1.6 architecture, TriCore 1.6 instructions are allowed
tc1.6.x	TriCore 1.6.x architecture, TriCore 1.6.x instructions are allowed
tc1.6.2	TriCore 1.6.2 architecture, TriCore 1.6.2 instructions are allowed

Default: **tc1.3**

Description

With this option you specify the core architecture for a target processor for which you create your application. If you use Eclipse or the control program, the TriCore toolset derives the core from the processor you selected. When the default core (**tc1.3**) is used, the define `__CORE_TC13__` is set to 1.

With **--core=tc1.3.1**, you can use TriCore 1.3.1 instructions in the assembly code. The define `__CORE_TC131__` is set to 1.

With **--core=tc1.6**, you can use TriCore 1.6 instructions in the assembly code. The define `__CORE_TC16__` is set to 1.

With **--core=tc1.6.x**, you can use TriCore 1.6.x instructions in the assembly code. The define `__CORE_TC16X__` is set to 1.

With **--core=tc1.6.2**, you can use TriCore 1.6.2 instructions in the assembly code. The define `__CORE_TC162__` is set to 1.

Example

To allow the use of TriCore 1.3.1 instructions in the assembly code, enter:

```
astc --core=tc1.3.1 test.src
```

Related information

Control program option **--cpu** (Select processor)

Assembler controls **$TC131**, **$TC16**, **$TC16X**, **$TC162**

Assembler option **--mmu-present** (MMU present)

Assembler option **--no-fpu** (Disable FPU instructions)

Assembler option: --debug-info (-g)

Menu entry

1. Select **Assembler » Symbols**.

2. Select an option from the **Generate symbolic debug** list.

Command line syntax

--debug-info[**=***flags*]

-g[*flags*]

You can set the following flags:

+/-asm	**a/A**	Assembly source line information
+/-hll	**h/H**	Pass high level language debug information (HLL)
+/-local	**l/L**	Assembler local symbols debug information
+/-smart	**s/S**	Smart debug information

Default: **--debug-info=+hll**

Default (without flags): **--debug-info=+smart**

Description

With this option you tell the assembler which kind of debug information to emit in the object file.

You cannot specify **--debug-info=+asm,+hll**. Either the assembler generates assembly source line information, or it passes HLL debug information.

When you specify **--debug-info=+smart**, the assembler selects which flags to use. If high level language information is available in the source file, the assembler passes this information (same as **--debug-info=-asm,+hll,-local**). If not, the assembler generates assembly source line information (same as **--debug-info=+asm,-hll,+local**).

With **--debug-info=AHLS** the assembler does not generate any debug information.

Related information

Assembler control **$DEBUG**

Assembler option: --define (-D)

Menu entry

1. Select **Assembler » Preprocessing**.

    The Defined symbols box right-below shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, demo=1)

    Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

--define=*macro_name*[**=***macro_definition*]

-D*macro_name*[**=***macro_definition*]

Description

With this option you can define a macro and specify it to the assembler preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the assembler with the option **--option-file** (**-f**) *file*.

Defining macros with this option (instead of in the assembly source) is, for example, useful in combination with conditional assembly as shown in the example below.

This option has the same effect as defining symbols via the .DEFINE, .SET, and .EQU directives (similar to #define in the C language). With the .MACRO directive you can define more complex macros.

Make sure you do not use a reserved keyword as a macro name, as this can lead to unexpected results.

Example

Consider the following assembly program with conditional code to assemble a demo program and a real program:

```
.IF DEMO == 1
...          ; instructions for demo application
.ELSE
...          ; instructions for the real application
.ENDIF
```

You can now use a macro definition to set the DEMO flag:

```
astc --define=DEMO test.src
astc --define=DEMO=1 test.src
```

Note that both invocations have the same effect.

Related information

Assembler option **--option-file** (Specify an option file)

Assembler option: --dep-file

Menu entry

-

Command line syntax

`--dep-file`[`=`*file*]

Description

With this option you tell the assembler to generate dependency lines that can be used in a Makefile. The dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension `.d`. When you specify a filename, all dependencies will be combined in the specified file.

Example

```
astc --dep-file=test.dep test.src
```

The assembler assembles the file `test.src`, which results in the output file `test.o`, and generates dependency lines in the file `test.dep`.

Related information

Assembler option **--make-target** (Specify target name for **--dep-file** output)

Assembler option: --diag

Menu entry

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

    The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

    A popup menu appears.

3. Select **Detailed Diagnostics Info**.

    A dialog box appears with additional information.

Command line syntax

--diag=[*format*:]{**all** | *nr*,...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

Example

To display an explanation of message number 244, enter:

```
astc --diag=244
```

This results in the following message and explanation:

```
W244: additional input files will be ignored
```

```
The assembler supports only a single input file. All other input files are ignored.
```

To write an explanation of all errors and warnings in HTML format to file `aserrors.html`, use redirection and enter:

```
astc --diag=html:all > aserrors.html
```

Related information

Section 6.6, *Assembler Error Messages*

Assembler option: --emit-locals

Menu entry

1. Select **Assembler » Symbols**.

2. Enable or disable one or both of the following options:

    *   Emit local EQU symbols

    *   Emit local non-EQU symbols

Command line syntax

--emit-locals[**=**_flag_,...]

You can set the following flags:

+/-equs	**e/E**	emit local EQU symbols
+/-symbols	**s/S**	emit local non-EQU symbols

Default: **--emit-locals=ES**

Default (without flags): **--emit-locals=+symbols**

Description

With the option **--emit-locals=+equs** the assembler also emits local EQU symbols to the object file. Normally, only global symbols and non-EQU local symbols are emitted. Having local symbols in the object file can be useful for debugging.

Related information

Assembler directive **.EQU**

Assembler option: --error-file

Menu entry

-

Command line syntax

--error-file[**=***file*]

Description

With this option the assembler redirects error messages to a file. If you do not specify a filename, the error file will be named after the output file with extension `.ers`.

Example

To write errors to `errors.ers` instead of `stderr`, enter:

```
astc --error-file=errors.ers test.src
```

Related information

Section 6.6, *Assembler Error Messages*

Assembler option: --error-limit

Menu entry

1. Select **Assembler » Diagnostics**.

2. Enter a value in the **Maximum number of emitted errors** field.

Command line syntax

```
--error-limit=number
```

Default: 42

Description

With this option you tell the assembler to only emit the specified maximum number of errors. When 0 (null) is specified, the assembler emits all errors. Without this option the maximum number of errors is 42.

Related information

Section 6.6, *Assembler Error Messages*

Assembler option: --help (-?)

Menu entry

-

Command line syntax

--help[**=***item*]

-?

You can specify the following arguments:

> **options**   Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

The following invocations all display a list of the available command line options:

```
astc -?
astc --help
astc
```

To see a detailed description of the available options, enter:

```
astc --help=options
```

Related information

-

Assembler option: --include-directory (-I)

Menu entry

1. Select **Assembler » Include Paths**.

   The Include paths box shows the directories that are added to the search path for include files.

2. To define a new directory for the search path, click on the **Add** button in the **Include paths** box.

3. Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

`--include-directory=`*path*`,...`

`-I`*path*`,...`

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory,

The order in which the assembler searches for include files is:

1. The pathname in the assembly file and the directory of the assembly source.

2. The path that is specified with this option.

3. The path that is specified in the environment variable `ASTCINC` when the product was installed.

4. The default directory `$(PRODDIR)\include`.

Example

Suppose that the assembly source file `test.src` contains the following lines:

`.INCLUDE 'myinc.inc'`

You can call the assembler as follows:

`astc --include-directory=c:\proj\include test.src`

First the assembler looks for the file `myinc.inc` in the directory where `test.src` is located. If it does not find the file, it looks in the directory `c:\proj\include` (this option). If the file is still not found, the assembler searches in the environment variable and then in the default include directory.

Related information

Assembler option **--include-file** (Include file at the start of the input file)

Assembler option: --include-file (-H)

Menu entry

1. Select **Assembler » Preprocessing**.

    The Pre-include files box shows the files that are currently included before the assembling starts.

2. To define a new file, click on the **Add** button in the **Pre-include files** box.

3. Type the full path and file name or select a file.

> Use the **Edit** and **Delete** button to change a file name or to remove a file from the list.

Command line syntax

`--include-file=`*file*`,...`

`-H`*file*`,...`

Description

With this option (set at project level) you include one extra file at the beginning of the assembly source file. The specified include file is included before all other includes. This is the same as specifying `.INCLUDE 'file'` at the beginning of your assembly source.

Example

```
astc --include-file=sfr/regtc1796b.def test.src
```

The file `regtc1796b.def` in the `sfr` subdirectory of the `include` directory is included at the beginning of `test.src` before it is assembled.

Related information

Assembler option **--include-directory** (Add directory to include file search path)

Assembler option: --kanji

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Enable the option **Allow Shift JIS Kanji in strings**.

Command line syntax

```
--kanji
```

Description

With this option you tell the assembler to support Shift JIS encoded Kanji multi-byte characters in strings. Without this option, encodings with 0x5c as the second byte conflict with the use of the backslash as an escape character. Shift JIS in comments is supported regardless of this option.

Note that Shift JIS also includes Katakana and Hiragana.

Related information

C compiler option **--language=+kanji** (Allow Shift JIS Kanji in strings)

Assembler option: --keep-output-files (-k)

Menu entry

Eclipse *always* removes the object file when errors occur during assembling.

Command line syntax

```
--keep-output-files
```

```
-k
```

Description

If an error occurs during assembling, the resulting object file (`.o`) may be incomplete or incorrect. With this option you keep the generated object file when an error occurs.

By default the assembler removes the generated object file when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated object. For example when you know that a particular error does not result in a corrupt object file.

Related information

Assembler option **--warnings-as-errors** (Treat warnings as errors)

Assembler option: --list-file (-l)

Menu entry

1. Select **Assembler » List File**.

2. Enable the option **Generate list file**.

3. Enable or disable the types of information to be included.

Command line syntax

--list-file[**=***file*]

-l[*file*]

Default: no list file is generated

Description

With this option you tell the assembler to generate a list file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

With the optional *file* you can specify an alternative name for the list file. By default, the name of the list file is the basename of the output file with the extension .lst.

Related information

Assembler option **--list-format** (Format list file)

Assembler option: --list-format (-L)

Menu entry

1. Select **Assembler » List File**.

2. Enable the option **Generate list file**.

3. Enable or disable the types of information to be included.

Command line syntax

--list-format=`flag`**,...**

-L`flags`

You can set the following flags:

+/-section	**d/D**	List section directives (`.SDECL`, `.SECT`)
+/-symbol	**e/E**	List symbol definition directives
+/-generic-expansion	**g/G**	List expansion of generic instructions
+/-generic	**i/I**	List generic instructions
+/-macro	**m/M**	List macro definitions
+/-empty-line	**n/N**	List empty source lines (newline)
+/-conditional	**p/P**	List conditional assembly
+/-equate	**q/Q**	List equate and set directives (`.EQU`, `.SET`)
+/-relocations	**r/R**	List relocations characters ('r')
+/-equate-values	**v/V**	List equate and set values
+/-wrap-lines	**w/W**	Wrap source lines
+/-macro-expansion	**x/X**	List macro expansions
+/-cycle-count	**y/Y**	List cycle counts
+/-define-expansion	**z/Z**	List define expansions

Use the following options for predefined sets of flags:

--list-format=0	**-L0**	All options disabled Alias for **--list-format=DEGIMNPQRVWXYZ**
--list-format=1	**-L1**	All options enabled Alias for **--list-format=degimnpqrvwxyz**

Default: **`--list-format=dEGiMnPqrVwXyZ`**

Description

With this option you specify which information you want to include in the list file.

On the command line you must use this option in combination with the option **--list-file** (**-l**).

Related information

Assembler option **--list-file** (Generate list file)

Assembler option **--section-info=+list** (Display section information in list file)

Assembler option: --make-target

Menu entry

-

Command line syntax

`--make-target=`*name*

Description

With this option you can overrule the default target name in the make dependencies generated by the option **--dep-file**. The default target name is the basename of the input file, with extension `.o`.

Example

```
astc --dep-file --make-target=../mytarget.o test.src
```

The assembler generates dependency lines with the default target name `../mytarget.o` instead of `test.o`.

Related information

Assembler option **--dep-file** (Generate dependencies in a file)

Assembler option: --mmu-present

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor Selection** list, select a processor that has an MMU.

Command line syntax

`--mmu-present`

Description

With this option you can use memory management instructions in the assembly code. When you select this option, the define __MMU__ is set to 1.

If you select a valid target processor, this option is automatically set, based on the chosen target processor.

Example

To allow the use of memory management unit (MMU) instructions in the assembly code, enter:

```
astc --mmu-present test.src
```

Related information

Control program option **--cpu** (Select processor)

Assembler control **$MMU**

Assembler option **--core** (Select TriCore architecture)

Assembler option **--no-fpu** (Disable FPU instructions)

Assembler option: --no-fpu

Menu entry

1.   Expand **C/C++ Build** and select **Processor**.

2.   From the **Processor Selection** list, select a processor that does not have an FPU.

Command line syntax

`--no-fpu`

Description

By default, you can use single precision floating-point (FPU) instructions in the assembly code. With this option you tell the assembler that FPU instructions are not allowed. When you select this option, the define `__FPU__` is set to 0.

If you select a valid target processor, this option is automatically set, based on the chosen target processor.

Example

To disable the use of floating-point unit (FPU) instructions in the assembly code, enter:

`astc --no-fpu test.src`

Related information

Control program option **--cpu** (Select processor)

Assembler control **$NO_FPU**

Assembler option **--core** (Select TriCore architecture)

Assembler option: --no-warnings (-w)

Menu entry

1. Select **Assembler » Diagnostics**.

   The Suppress warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.

3. Enter the numbers, separated by commas, of the warnings you want to suppress (for example `201,202`). Or you can use the **Add** button multiple times.

4. To suppress all warnings, enable the option **Suppress all warnings**.

Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

Command line syntax

--no-warnings[**=***number*,...]

-w[*number*,...]

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.

- If you specify this option but without numbers, all warnings are suppressed.

- If you specify this option with a number, only the specified warning is suppressed. You can specify the option **--no-warnings=***number* multiple times.

Example

To suppress warnings 201 and 202, enter:

```
astc test.src --no-warnings=201,202
```

Related information

Assembler option **--warnings-as-errors** (Treat warnings as errors)

Assembler option: --optimize (-O)

Menu entry

1. Select **Assembler » Optimization**.

2. Select one or more of the following options:

    *   Optimize generic instructions

    *   Optimize instruction size

Command line syntax

`--optimize=`*flag*`,...`

`-O`*flags*

You can set the following flags:

+/-generics	**g/G**	Allow generic instructions
+/-instr-size	**s/S**	Optimize instruction size

Default: `--optimize=gs`

Description

With this option you can control the level of optimization. For details about each optimization see Section 6.4, *Assembler Optimizations*.

Related information

Assembler control **$HW_ONLY**

Section 6.4, *Assembler Optimizations*

Assembler option: --option-file (-f)

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--option-file** to the **Additional options** field.

    Be aware that the options in the option file are added to the assembler options you have set in the other pages. Only in extraordinary cases you may want to use them in combination.

Command line syntax

--option-file=*file*,...

-f *file*,...

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the assembler.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

Option files can also be generated on the fly, for example by the make utility. You can specify the option **--option-file** multiple times.

Format of an option file

*   Multiple arguments on one line in the option file are allowed.

*   To include whitespace in an argument, surround the argument with single or double quotes.

*   If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

    ```
    "This has a single quote ' embedded"

    'This has a double quote " embedded'

    'This has a double quote " and a single quote '"' embedded"
    ```

*   When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

    ```
    "This is a continuation \
    line"

            -> "This is a continuation line"
    ```

- It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
--debug=+asm,-local
test.src
```

Specify the option file to the assembler:

```
astc --option-file=myoptions
```

This is equivalent to the following command line:

```
astc --debug=+asm,-local test.src
```

Related information

-

Assembler option: --output (-o)

Menu entry

Eclipse names the output file always after the input file.

Command line syntax

--output=file

-o file

Description

With this option you can specify another filename for the output file of the assembler. Without this option, the basename of the assembly source file is used with extension `.o`.

Example

To create the file `relobj.o` instead of `asm.o`, enter:

```
astc --output=relobj.o asm.src
```

Related information

-

Assembler option: --page-length

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--page-length** to the **Additional options** field.

Command line syntax

--page-length=*number*

Default: 72

Description

If you generate a list file with the assembler option **--list-file**, this option sets the number of lines in a page in the list file. The default is 72, the minimum is 10. As a special case, a page length of 0 turns off page breaks.

Related information

Assembler option **--list-file** (Generate list file)

Assembler control **$PAGE**

Assembler option: --page-width

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--page-width** to the **Additional options** field.

Command line syntax

`--page-width=`*number*

Default: 132

Description

If you generate a list file with the assembler option **--list-file**, this option sets the number of columns per line on a page in the list file. The default is 132, the minimum is 40.

Related information

Assembler option **--list-file** (Generate list file)

Assembler control **$PAGE**

Assembler option: --preprocess (-E)

Menu entry

-

Command line syntax

`--preprocess`

`-E`

Description

With this option the assembler will only preprocess the assembly source file. The assembler sends the preprocessed file to stdout.

Related information

-

Assembler option: --preprocessor-type (-m)

Menu entry

-

Command line syntax

--preprocessor-type=$type$

-m$type$

You can set the following preprocessor types:

none	**n**	No preprocessor
tasking	**t**	TASKING preprocessor

Default: **--preprocessor-type=tasking**

Description

With this option you select the preprocessor that the assembler will use. By default, the assembler uses the TASKING preprocessor.

When the assembly source file does not contain any preprocessor symbols, you can specify to the assembler not to use a preprocessor.

Related information

-

Assembler option: --require-end

Menu entry

-

Command line syntax

`--require-end`

Description

With this option the assembly source must be terminated with a `.END` directive.

Related information

`.END` (End of assembly)

Assembler option: --section-info (-t)

Menu entry

1.   Select **Assembler » List File**.

2.   Enable the option **Generate list file**.

3.   Enable the option **List section summary**.

and/or

1.   Select **Assembler » Diagnostics**.

2.   Enable the option **Display section summary**.

Command line syntax

--section-info[**=***flag*,...]

-t[*flags*]

You can set the following flags:

+/-console	**c/C**	Display section summary on console
+/-list	**l/L**	List section summary in list file

Default: **--section-info=CL**

Default (without flags): **--section-info=cl**

Description

With this option you tell the assembler to display section information. For each section its memory space, size, total cycle counts and name is listed on stdout and/or in the list file.

The cycle count consists of two parts: the total accumulated count for the section and the total accumulated count for all repeated instructions. In the case of nested loops it is possible that the total supersedes the section total.

Example

To writes the section information to the list file and also display the section information on stdout, enter:

```
astc --list-file --section-info asm.src
```

Related information

Assembler option **--list-file** (Generate list file)

Assembler option: --silicon-bug

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor selection** list, select a processor.

    *The **CPU problem bypasses and checks** box shows the available workarounds/checks available for the selected processor.*

3. (Optional) Select **Show all CPU problem bypasses and checks**.

4. Click **Select All** or select one or more individual options.

Command line syntax

`--silicon-bug=`*arg*`,...`

You can give one or more of the following arguments:

all-tc1130	All TriCore TC1130 checks
all-tc1164	All TriCore TC1164 checks
all-tc1166	All TriCore TC1166 checks
all-tc131	All TriCore 1.3.1 checks
all-tc1736	All TriCore TC1736 checks
all-tc1762	All TriCore TC1762 checks
all-tc1764	All TriCore TC1764 checks
all-tc1766	All TriCore TC1766 checks
all-tc1767	All TriCore TC1767 checks
all-tc1792	All TriCore TC1792 checks
all-tc1796	All TriCore TC1796 checks
all-tc1797	All TriCore TC1797 checks
cpu-tc013	Check for CPU_TC.013
cpu-tc048	Check for CPU_TC.048
cpu-tc060	Check for CPU_TC.060
cpu-tc065	Check for CPU_TC.065
cpu-tc068	Check for CPU_TC.068
cpu-tc069	Check for CPU_TC.069
cpu-tc070	Check for CPU_TC.070
cpu-tc071	Check for CPU_TC.071
cpu-tc072	Check for CPU_TC.072

cpu-tc074	Workaround for CPU_TC.074
cpu-tc081	Check for CPU_TC.081
cpu-tc082	Check for CPU_TC.082
cpu-tc083	Check for CPU_TC.083
cpu-tc094	Check for CPU_TC.094
cpu-tc095	Check for CPU_TC.095
cpu-tc096	Check for CPU_TC.096
cpu-tc104	Check for CPU_TC.104
cpu-tc105	Check for CPU_TC.105
cpu-tc106	Check for CPU_TC.106
cpu-tc108	Check for CPU_TC.108
cpu-tc109	Check for CPU_TC.109
cpu-tc116	Check for CPU_TC.116

Description

With this option you specify for which hardware problems the assembler should check or generate workarounds. Please refer to Chapter 18, *CPU Problem Bypasses and Checks* for more information about the individual problems and workarounds.

Instead of this option you can use the following assembler controls:

```
$CPU_TCnum {ON|OFF}
```

When you use this option or control, the define __CPU_TC*num*__ is set to 1.

Example

To check for problems CPU_TC.013 and CPU_TC.048, enter:

```
astc --silicon-bug=cpu-tc013,cpu-tc048 test.src
```

Related information

Chapter 18, *CPU Problem Bypasses and Checks*

Assembler controls **$CPU_TC***num*

C compiler option **--silicon-bug**

Assembler option: --symbol-scope (-i)

Menu entry

1. Select **Assembler » Symbols**.

2. Enable or disable the option **Set default symbol scope to global**.

Command line syntax

`--symbol-scope=`*scope*

`-i`*scope*

You can set the following scope:

global	**g**	Default symbol scope is global
local	**l**	Default symbol scope is local

Default: `--symbol-scope=local`

Description

With this option you tell the assembler how to treat symbols that you have not specified explicitly as global or local. By default the assembler treats all symbols as local symbols unless you have defined them explicitly as global.

Related information

Assembler directive **.GLOBAL**

Assembler directive **.LOCAL**

Assembler control **$IDENT**

Assembler option: --user-mode

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--user-mode** to the **Additional options** field.

Command line syntax

--user-mode=*mode*

You can specify the following *mode* arguments:

user-0	User-0 unprivileged mode
user-1	User-1 privileged mode
kernel	Kernel/Supervisor mode

Default: **--user-mode=kernel**

Description

With this option you specify the mode (I/O privilege mode) the TriCore runs in: User-0, User-1 or Kernel/Supervisor. The availability of some instructions depends on this mode. Most instructions run in all modes. The instructions `enable` and `disable` run in User-1 or kernel mode only. The instructions `bisr`, `mtcr`, `cachea.i` and `tlb` instructions run in kernel mode only.

The define `__UM_KERNEL__` or `__UM_USER_1__` is set to 1, depending on the mode.

Example

To restrict the instructions in the assembly code to User-1 mode, enter:

```
astc --user-mode=user-1 test.src
```

Related information

C compiler option **--user-mode** (Select user mode)

Assembler option: --version (-V)

Menu entry

-

Command line syntax

--version

-V

Description

Display version information. The assembler ignores all other options or input files.

Example

```
astc --version
```

The assembler does not assemble any files but displays the following version information:

```
TASKING VX-toolset for TriCore: assembler    vx.yrz Build nnn
Copyright 2002-year Altium BV                Serial# 00000000
```

Related information

-

Assembler option: --warn-access-interrupt-enable

Menu entry

-

Command line syntax

`--warn-access-interrupt-enable`

Description

With this option the assembler gives a warning when the instructions `disable`, `disable D[a]`, `enable` or `restore D[a]` are used. These instructions have a high potential for deadlocks in a multi-core environment.

Related information

-

Assembler option: --warnings-as-errors

Menu entry

1. Select **Global Options**.

2. Enable the option **Treat warnings as errors**.

Command line syntax

`--warnings-as-errors`[`=`*number*`,...`]

Description

If the assembler encounters an error, it stops assembling. When you use this option without arguments, you tell the assembler to treat all warnings as errors. This means that the exit status of the assembler will be non-zero after one or more assembler warnings. As a consequence, the assembler now also stops after encountering a warning.

You can limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

Assembler option **--no-warnings** (Suppress some or all warnings)

10.5. Linker Options

This section lists all linker options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the linker via the control program. Therefore, it uses the syntax of the control program to pass options and files to the linker. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Miscellaneous**.

4. In the **Additional options** field, enter one or more command line options.

    *Because Eclipse uses the control program, Eclipse automatically precedes the option with **-Wl** to pass the option via the control program directly to the linker.*

Be aware that some command line options are not useful in Eclipse or just do not have any effect. For example, the option **--keep-output-files** keeps files after an error occurred. When you specify this option in Eclipse, it will have no effect because Eclipse always removes the output file after an error had occurred.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+***longflag*. To switch a flag off, use an uppercase letter or a **-***longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
ltc -mfkl test.o
ltc --map-file-format=+files,+link,+locate test.o
```

When you do not specify an option, a default value may become active.

Linker option: --auto-base-register

Menu entry

1. Select **Linker » Miscellaneous**.

2. Add the option **--auto-base-register** to the **Additional options** field.

Command line syntax

```
--auto-base-register
```

Description

By default, the linker groups sections that belong to A0, A1, A8 or A9 each to their own group, as specified in the LSL file `base_address_groups.lsl`. When a base register addressed section comes out of range for its base register as assigned in the object file, the linker by default issues an error message. With option **--auto-base-register** the linker tries to find a different base address register that is in range. The search order is A0, A1, A8, A9. If the section is not in range of any of the base address registers, the linker still issues an error message.

This behavior makes it possible to place a section in a different base addressable group than assigned at compile time (with `__a0`, `__a1`, `__a8`, `__a9`). For example a `.data_a8` section can be grouped with the group for the A9 base address register.

Related information

Section 7.8.4, *Structure of a Linker Script File*

Linker option: --case-insensitive

Menu entry

1. Select **Linker » Miscellaneous**.

2. Enable the option **Link case insensitive**.

Command line syntax

```
--case-insensitive
```

Default: case sensitive

Description

With this option you tell the linker not to distinguish between uppercase and lowercase characters in symbols. By default the linker considers uppercase and lowercase characters as different characters.

Assembly source files that are generated by the compiler must *always* be assembled and thus linked case sensitive. When you have written your own assembly code and specified to assemble it case insensitive, you must also link the `.o` file case insensitive.

Related information

Assembler option **--case-insensitive**

Linker option: --chip-output (-c)

Menu entry

1. Select **Linker » Output Format**.

2. Enable the option **Generate Intel Hex format file** and/or **Generate S-records file**.

3. Enable the option **Create file for each memory chip**.

4. Optionally, specify the **Size of addresses**.

    Eclipse always uses the project name as the basename for the output file.

Command line syntax

--chip-output=[*basename*]**:***format*[**:***addr_size*]**,**...

-c[*basename*]**:***format*[**:***addr_size*]**,**...

You can specify the following formats:

IHEX	Intel Hex
SREC	Motorola S-records

The *addr_size* specifies the size of the addresses in bytes (record length). For Intel Hex you can use the values **1**, **2** or **4** bytes (default). For Motorola-S you can specify: **2** (S1 records), **3** (S2 records) or **4** bytes (S3 records, default).

Description

With this option you specify the Intel Hex or Motorola S-record output format for loading into a PROM-programmer. The linker generates a file for each ROM memory defined in the LSL file, where sections are located:

```
memory memname
{  type=rom;  }
```

The name of the file is the name of the Eclipse project or, on the command line, the name of the memory device that was emitted with extension `.hex` or `.sre`. Optionally, you can specify a *basename* which prepends the generated file name.

> The linker always outputs a debugging file in ELF/DWARF format and optionally an absolute object file in Intel Hex-format and/or Motorola S-record format.

Example

To generate Intel Hex output files for each defined memory, enter the following on the command line:

```
ltc --chip-output=myfile:IHEX test1.o
```

In this case, this generates the file `myfile_`*memname*`.hex`.

Related information

Linker option **--output** (Output file)

Linker option: --core (-C)

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor Selection** list, select a processor or select **User defined TriCore ...**.

3. From the **Multi-core configuration** list, select a TriCore single-core or multi-core.

Command line syntax

--core=`TriCore-core`

-C`TriCore-core`

You can specify the following TriCore cores:

mpe:tc0	TriCore core 0
mpe:vtc	TriCore multi-core

Default: **mpe:vtc**

Description

With this option you specify the core for the target processor for which you create your application.

The core is specified as `mpe:tc0` for single-core or `mpe:vtc` for multi-core. For example, the file `tc27x.lsl` in the `include.lsl` directory, contains a description of derivative `tc27x` and the supported TriCore cores and MCS cores. `mpe` is the multi-processor environment as specified in the LSL file. The file `tc27x_tc0.lsl` contains a description for the single TriCore core `tc0`.

Example

To link objects for the TriCore core `mpe:tc0`, enter:

```
ltc  -o test.elf -dtc27x_tc0.lsl -dextmem.lsl
    -D__CPU__=tc27x -D__PROC_TC27X__
    --core=mpe:tc0 --map-file test.o
    -lcs_fpu -lfp_fpu -lrt -L~/ctc/lib/tc16x
```

Related information

Control program option **--lsl-core** (Specify LSL core)

Linker option: --define (-D)

Menu entry

1. Select **Linker » Script File**.

    The Defined symbols box shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, demo=1)

Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

--define=*macro_name*[**=***macro_definition*]

-D*macro_name*[**=***macro_definition*]

Description

With this option you can define a macro and specify it to the linker LSL file preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like; just use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the linker with the option **--option-file** (**-f**) *file*.

The definition can be tested by the preprocessor with #if, #ifdef and #ifndef, for conditional locating.

Make sure you do not use a reserved keyword as a macro name, as this can lead to unexpected results.

Example

To define the RESET vector, which is used in the linker script file tc1v1_3.lsl, enter:

```
ltc test.o -otest.elf --lsl-file=tc1v1_3.lsl --define=RESET=0xa0000000
```

Related information

Linker option **--option-file** (Specify an option file)

Linker option: --diag

Menu entry

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

    The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

    A popup menu appears.

3. Select **Detailed Diagnostics Info**.

    A dialog box appears with additional information.

Command line syntax

--diag=[*format*:]{**all** | *nr*,...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the linker does not link/locate any files.

Example

To display an explanation of message number 106, enter:

```
ltc --diag=106
```

This results in the following message and explanation:

```
E106: unresolved external: <message>
```

```
The linker could not resolve all external symbols.
```

```
This is an error when the incremental linking option is disabled.
The <message> indicates the symbol that is unresolved.
```

To write an explanation of all errors and warnings in HTML format to file `lkerrors.html`, use redirection and enter:

```
ltc --diag=html:all > lkerrors.html
```

Related information

Section 7.11, *Linker Error Messages*

Linker option: --error-file

Menu entry

-

Command line syntax

--error-file[**=***file*]

Description

With this option the linker redirects error messages to a file. If you do not specify a filename, the error file is `ltc.elk`.

Example

To write errors to `errors.elk` instead of `stderr`, enter:

```
ltc --error-file=errors.elk test.o
```

Related information

Section 7.11, *Linker Error Messages*

Linker option: --error-limit

Menu entry

1. Select **Linker » Diagnostics**.

2. Enter a value in the **Maximum number of emitted errors** field.

Command line syntax

`--error-limit=`*number*

Default: 42

Description

With this option you tell the linker to only emit the specified maximum number of errors. When 0 (null) is specified, the linker emits all errors. Without this option the maximum number of errors is 42.

Related information

Section 7.11, *Linker Error Messages*

Linker option: --extern (-e)

Menu entry

-

Command line syntax

--extern=`symbol`**,...**

-e`symbol`**,...**

Description

With this option you force the linker to consider the given symbol as an undefined reference. The linker tries to resolve this symbol, either the symbol is defined in an object file or the linker extracts the corresponding symbol definition from a library.

This option is, for example, useful if the startup code is part of a library. Because your own application does not refer to the startup code, you can force the startup code to be extracted by specifying the symbol _START as an unresolved external.

Example

Consider the following invocation:

```
ltc mylib.a
```

Nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through `mylib.a`.

```
ltc --extern=_START mylib.a
```

In this case the linker searches for the symbol _START in the library and (if found) extracts the object that contains _START, the startup code. If this module contains new unresolved symbols, the linker looks again in `mylib.a`. This process repeats until no new unresolved symbols are found.

Related information

Section 7.3, *Linking with Libraries*

Linker option: --first-library-first

Menu entry

-

Command line syntax

`--first-library-first`

Description

When the linker processes a library it searches for symbols that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference the linker extracts the object that contains the definition from the library.

By default the linker processes object files and libraries in the order in which they appear on the command line. If you specify the option **--first-library-first** the linker always tries to take the symbol definition from the library that appears first on the command line before scanning subsequent libraries.

This is for example useful when you are working with a newer version of a library that partially overlaps the older version. Because they do not contain exactly the same functions, you have to link them both. However, when a function is present in both libraries, you may want the linker to extract the most recent function.

Example

Consider the following example:

```
ltc --first-library-first a.a test.o b.a
```

If the file `test.o` calls a function which is both present in `a.a` and `b.a`, normally the function in `b.a` would be extracted. With this option the linker first tries to extract the symbol from the first library `a.a`.

Note that routines in `b.a` that call other routines that are present in both `a.a` and `b.a` are now also resolved from `a.a`.

Related information

Linker option **--no-rescan** (Rescan libraries to solve unresolved externals)

Linker option: --global-type-checking

Menu entry

-

Command line syntax

`--global-type-checking`

Description

Use this option when you want the linker to check the types of variable and function references against their definitions, using DWARF 2 or DWARF 3 debug information.

This check should give the same result as the C compiler when you use MIL linking.

Related information

-

Linker option: --help (-?)

Menu entry

-

Command line syntax

--help[**=**_item_]

-?

You can specify the following arguments:

    options   Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

The following invocations all display a list of the available command line options:

```
ltc -?
ltc --help
ltc
```

To see a detailed description of the available options, enter:

```
ltc --help=options
```

Related information

-

Linker option: --hex-format

Menu entry

1. Select **Linker » Output Format**.

2. Enable the option **Generate Intel Hex format file**.

3. Enable or disable the option **Emit start address record**.

Command line syntax

`--hex-format=`*flag*`,...`

You can set the following flag:

+/-start-address   **s/S**   Emit start address record

Default: `--hex-format=s`

Description

With this option you can specify to emit or omit the start address record from the hex file.

Related information

Linker option **--output** (Output file)

Section 15.2, *Intel Hex Record Format*

Linker option: --hex-record-size

Menu entry

1. Select **Linker » Output Format**.

2. Enable the option **Generate Intel Hex format file**.

3. Select **Linker » Miscellaneous**.

4. Add the option **--hex-record-size** to the **Additional options** field.

Command line syntax

`--hex-record-size=`*size*

Default: 32

Description

With this option you can set the size (width) of the Intel Hex data records.

Related information

Linker option **--output** (Output file)

Section 15.2, *Intel Hex Record Format*

Linker option: --import-object

Menu entry

1. Select **Linker » Data Objects**.

    The Data objects box shows the list of object files that are imported.

2. To add a data object, click on the **Add** button in the **Data objects** box.

3. Type or select a binary file (including its path).

Use the **Edit** and **Delete** button to change a filename or to remove a data object from the list.

Command line syntax

```
--import-object=file,...
```

Description

With this option the linker imports a binary *file* containing raw data and places it in a section. The section name is derived from the filename, in which dots are replaced by an underscore. So, when importing a file called `my.jpg`, a section with the name `my_jpg` is created. In your application you can refer to the created section by using linker labels.

Related information

Section 7.6, *Importing Binary Files*

Linker option: --include-directory (-I)

Menu entry

-

Command line syntax

--include-directory=*path*,...

-I*path*,...

Description

With this option you can specify the path where your LSL include files are located. A relative path will be relative to the current directory.

The order in which the linker searches for LSL include files is:

1. The pathname in the LSL file and the directory where the LSL file is located (only for #include files that are enclosed in "")

2. The path that is specified with this option.

3. The default directory $(PRODDIR)\include.lsl.

Example

Suppose that your linker script file mylsl.lsl contains the following line:

```
#include "myinc.inc"
```

You can call the linker as follows:

```
ltc --include-directory=c:\proj\include --lsl-file=mylsl.lsl test.o
```

First the linker looks for the file myinc.inc in the directory where mylsl.lsl is located. If it does not find the file, it looks in the directory c:\proj\include (this option). Finally it looks in the directory $(PRODDIR)\include.lsl.

Related information

Linker option **--lsl-file** (Specify linker script file)

Linker option: --incremental (-r)

Menu entry

-

Command line syntax

```
--incremental
```

```
-r
```

Description

Normally the linker links and locates the specified object files. With this option you tell the linker only to link the specified files. The linker creates a linker output file `.out`. You then can link this file again with other object files until you have reached the final linker output file that is ready for locating.

In the last pass, you call the linker without this option with the final linker output file `.out`. The linker will now locate the file.

Example

In this example, the files `test1.o`, `test2.o` and `test3.o` are incrementally linked:

1. `ltc --incremental test1.o test2.o --output=test.out`

   test1.o and test2.o are linked

2. `ltc --incremental test3.o test.out`

   test3.o and test.out are linked, task1.out is created

3. `ltc task1.out`

   task1.out is located

Related information

Section 7.4, *Incremental Linking*

Linker option: --keep-output-files (-k)

Menu entry

Eclipse *always* removes the output files when errors occurred.

Command line syntax

```
--keep-output-files
```

```
-k
```

Description

If an error occurs during linking, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the linker removes the generated output file when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated file. For example when you know that a particular error does not result in a corrupt object file, or when you want to inspect the output file, or send it to Altium support.

Related information

Linker option **--warnings-as-errors** (Treat warnings as errors)

Linker option: --library (-l)

Menu entry

1. Select **Linker » Libraries**.

    The Libraries box shows the list of libraries that are linked with the project.

2. To add a library, click on the **Add** button in the **Libraries** box.

3. Type or select a library (including its path).

4. Optionally, disable the option **Link default libraries**.

Use the **Edit** and **Delete** button to change a library name or to remove a library from the list.

Command line syntax

--library=_name_

-l_name_

Description

With this option you tell the linker to use system library lib*name*.a, where *name* is a string. The linker first searches for system libraries in any directories specified with **--library-directory**, then in the directories specified with the environment variables `LIBTC1V1_3` / `LIBTC1V1_3_1` / `LIBTC1V1_6` / `LIBTC1V1_6_X` / `LIBTC1V1_6_2`, unless you used the option **--ignore-default-library-path**.

Example

To search in the system library `libc.a` (C library):

```
ltc test.o mylib.a --library=c
```

The linker links the file `test.o` and first looks in library `mylib.a` (in the current directory only), then in the system library `libc.a` to resolve unresolved symbols.

Related information

Linker option **--library-directory** (Additional search path for system libraries)

Section 7.3, *Linking with Libraries*

Linker option: --library-directory (-L) / --ignore-default-library-path

Menu entry

1. Select **Linker » Libraries**.

   The Library search path box shows the directories that are added to the search path for library files.

2. To define a new directory for the search path, click on the **Add** button in the **Library search path** box.

3. Type or select a path.

   Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

```
--library-directory=path,...
-Lpath,...

--ignore-default-library-path
-L
```

Description

With this option you can specify the path(s) where your system libraries, specified with the option **--library** (**-l**), are located. If you want to specify multiple paths, use the option **--library-directory** for each separate path.

The default path is
`$(PRODDIR)\lib\[tc13][tc131][tc13_mmu][tc16][tc16_mmu][tc16x][tc162].`

If you specify only **-L** (without a pathname) or the long option **--ignore-default-library-path**, the linker will not search the default path and also not in the paths specified in the environment variables `LIBTC1V1_3 / LIBTC1V1_3_1 / LIBTC1V1_6 / LIBTC1V1_6_X / LIBTC1V1_6_2`. So, the linker ignores steps 2 and 3 as listed below.

The priority order in which the linker searches for system libraries specified with the option **--library** (**-l**) is:

1. The path that is specified with the option **--library-directory**.

2. The path that is specified in the environment variables `LIBTC1V1_3 / LIBTC1V1_3_1 / LIBTC1V1_6 / LIBTC1V1_6_X / LIBTC1V1_6_2`.

3. The default directory
   `$(PRODDIR)\lib\[tc13][tc131][tc13_mmu][tc16][tc16_mmu][tc16x][tc162].`

Example

Suppose you call the linker as follows:

```
ltc test.o --library-directory=c:\mylibs --library=c
```

First the linker looks in the directory `c:\mylibs` for library `libc.a` (this option). If it does not find the requested libraries, it looks in the directory that is set with the environment variables `LIBTC1V1_3 / LIBTC1V1_3_1 / LIBTC1V1_6 / LIBTC1V1_6_X / LIBTC1V1_6_2`. Then the linker looks in the default directory
`$(PRODDIR)\lib\[tc13][tc131][tc13_mmu][tc16][tc16_mmu][tc16x][tc162]` for libraries.

Related information

Linker option **--library** (Link system library)

Section 7.3.1, *How the Linker Searches Libraries*

Linker option: --link-only

Menu entry

-

Command line syntax

```
--link-only
```

Description

With this option you suppress the locating phase. The linker stops after linking and informs you about unresolved references.

Related information

Control program option **--create=relocatable** (**-cl**) (Stop after linking)

Linker option: --lsl-check

Menu entry

-

Command line syntax

```
--lsl-check
```

Description

With this option the linker just checks the syntax of the LSL file(s) and exits. No linking or locating is performed. Use the option **--lsl-file** to specify the name of the Linker Script File you want to test.

Related information

Linker option **--lsl-file** (Linker script file)

Linker option **--lsl-dump** (Dump LSL info)

Section 7.8, *Controlling the Linker with a Script*

Linker option: --lsl-dump

Menu entry

-

Command line syntax

--lsl-dump[=*file*]

Description

With this option you tell the linker to dump the LSL part of the map file in a separate file, independent of the option **--map-file** (generate map file). If you do not specify a filename, the file `ltc.ldf` is used.

Related information

Linker option **--map-file-format** (Map file formatting)

Linker option: --lsl-file (-d)

Menu entry

An LSL file can be generated when you create your project in Eclipse:

1. From the **File** menu, select **File » New » TASKING TriCore C/C++ Project**.

   The New C/C++ Project wizard appears.

2. Fill in the project settings in each dialog and click **Next >** until the **TriCore Project Settings** appear.

3. Enable the option **Add linker script file to the project** and click **Finish**.

   Eclipse creates your project and the file project.lsl in the project directory.

The LSL file can be specified in the Properties dialog:

1. Select **Linker » Script File**.

2. Specify a LSL file in the **Linker script file (.lsl)** field (default `../${ProjName}.lsl`).

Command line syntax

--lsl-file=`file`

-d`file`

Description

A linker script file contains vital information about the core for the locating phase of the linker. A linker script file is coded in LSL and contains the following types of information:

• the architecture definition describes the core's hardware architecture.

• the memory definition describes the physical memory available in the system.

• the section layout definition describes how to locate sections in memory.

With this option you specify a linker script file to the linker. If you do not specify this option, the linker uses a default script file. You can specify the existing file *target*.lsl or the name of a manually written linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

Related information

Linker option **--lsl-check** (Check LSL file(s) and exit)

Section 7.8, *Controlling the Linker with a Script*

Linker option: --map-file (-M)

Menu entry

1. Select **Linker » Map File**.

2. Enable the option **Generate XML map file format (.mapxml) for map file viewer**.

3. (Optional) Enable the option **Generate map file**.

4. Enable or disable the types of information to be included.

Command line syntax

--map-file[**=***file*][**:XML**]

-M[*file*][**:XML**]

Default (Eclipse): XML map file is generated

Default (linker): no map file is generated

Description

With this option you tell the linker to generate a linker map file. If you do not specify a filename and you specified the option **--output**, the linker uses the same basename as the output file with the extension .map. If you did not specify the option **--output**, the linker uses the file task1.map. Eclipse names the .map file after the project.

In Eclipse the XML variant of the map file (extension .mapxml) is used for graphical display in the map file viewer.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (.o) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

Related information

Linker option **--map-file-format** (Format map file)

Section 14.2, *Linker Map File Format*

Linker option: --map-file-format (-m)

Menu entry

1. Select **Linker » Map File**.

2. Enable the option **Generate XML map file format (.mapxml) for map file viewer**.

3. (Optional) Enable the option **Generate map file**.

4. Enable or disable the types of information to be included.

Command line syntax

```
--map-file-format=flag,...
```

```
-mflags
```

You can set the following flags:

+/-callgraph	**c/C**	Include call graph information
+/-removed	**d/D**	Include information on removed sections
+/-files	**f/F**	Include processed files information
+/-invocation	**i/I**	Include information on invocation and tools
+/-link	**k/K**	Include link result information
+/-locate	**l/L**	Include locate result information
+/-memory	**m/M**	Include memory usage information
+/-nonalloc	**n/N**	Include information of non-alloc sections
+/-overlay	**o/O**	Include overlay information
+/-statics	**q/Q**	Include module local symbols information
+/-crossref	**r/R**	Include cross references information
+/-lsl	**s/S**	Include processor and memory information
+/-rules	**u/U**	Include locate rules

Use the following options for predefined sets of flags:

--map-file-format=0	**-m0**	Link information Alias for **-mcDfikLMNoQrSU**
--map-file-format=1	**-m1**	Locate information Alias for **-mCDfiKlMNoQRSU**
--map-file-format=2	**-m2**	Most information Alias for **-mcdfiklmNoQrSu**

Default: `--map-file-format=2`

Description

With this option you specify which information you want to include in the map file.

On the command line you must use this option in combination with the option **--map-file** (**-M**).

Related information

Linker option **--map-file** (Generate map file)

Section 14.2, *Linker Map File Format*

Linker option: --misra-c-report

Menu entry

-

Command line syntax

--misra-c-report[**=***file*]

Description

With this option you tell the linker to create a MISRA C Quality Assurance report. This report lists the various modules in the project with the respective MISRA C settings at the time of compilation. If you do not specify a filename, the file *basename*.mcr is used.

Related information

C compiler option **--misrac** (MISRA C checking)

Linker option: --munch

Menu entry

-

Command line syntax

`--munch`

Description

With this option you tell the linker to activate the muncher in the pre-locate phase.

The muncher phase is a special part of the linker that creates sections containing a list of pointers to the initialization and termination routines. The list of pointers is consulted at run-time by startup code invoked from `main`, and the routines on the list are invoked at the appropriate times.

Related information

-

Linker option: --new-task

Menu entry

-

Command line syntax

`--new-task`

Description

With this option the linker creates an additional task. Any options that follow only apply to the new task.

The linker processes options on the command line from left to right. To know whether a certain option belongs to a different task it uses this option. This implies that all options for a given task must be fully specified before moving on to the next.

Related information

-

Linker option: --non-romable

Menu entry

-

Command line syntax

`--non-romable`

Description

With this option you tell the linker that the application must not be located in ROM. The linker will locate all ROM sections, including a copy table if present, in RAM. When the application is started, the data sections are re-initialized and the BSS sections are cleared as usual.

This option is, for example, useful when you want to test the application in RAM before you put the final application in ROM. This saves you the time of flashing the application in ROM over and over again.

If you want to locate your application in RAM only, without using ROM/flash resources of the chip, for example when you run the debugger in RAM only, also specify the options **--no-rom-copy** and **--user-provided-initialization-code**.

Related information

Linker option **--no-rom-copy** (Do not generate ROM copy)

Linker option **--user-provided-initialization-code** (Own initialization code, no standard copy table)

Linker option: --no-rescan

Menu entry

1. Select **Linker » Libraries**.

2. Disable the option **Rescan libraries to solve unresolved externals**.

Command line syntax

```
--no-rescan
```

Description

When the linker processes a library it searches for symbol definitions that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference the linker extracts the object that contains the definition from the library. The linker processes object files and libraries in the order in which they appear on the command line.

When all objects and libraries are processed the linker checks if there are unresolved symbols left. If so, the default behavior of the linker is to rescan all libraries in the order given at the command line. The linker stops rescanning the libraries when all symbols are resolved, or when the linker could not resolve any symbol(s) during the rescan of all libraries. Notice that resolving one symbol may introduce new unresolved symbols.

With this option, you tell the linker to scan the object files and libraries only once. When the linker has not resolved all symbols after the first scan, it reports which symbols are still unresolved. This option is useful if you are building your own libraries. The libraries are most efficiently organized if the linker needs only one pass to resolve all symbols.

Related information

Linker option **--first-library-first** (Scan libraries in given order)

Linker option: --no-rom-copy (-N)

Menu entry

-

Command line syntax

`--no-rom-copy`

`-N`

Description

With this option the linker will not generate a ROM copy for data sections. A copy table is generated and contains entries to clear BSS sections. However, no entries to copy data sections from ROM to RAM are placed in the copy table.

The data sections are initialized when the application is downloaded. The data sections are not re-initialized when the application is restarted.

Related information

Linker option **--non-romable** (Application is not romable)

Linker option **--user-provided-initialization-code** (Own initialization code, no standard copy table)

Linker option: --no-warnings (-w)

Menu entry

1. Select **Linker » Diagnostics**.

    The Suppress warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.

3. Enter the numbers, separated by commas, of the warnings you want to suppress (for example `135,136`). Or you can use the **Add** button multiple times.

4. To suppress all warnings, enable the option **Suppress all warnings**.

Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

Command line syntax

`--no-warnings``[=`*`number`*`,...]`

`-w``[`*`number`*`,...]`

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

*   If you do not specify this option, all warnings are reported.

*   If you specify this option but without numbers, all warnings are suppressed.

*   If you specify this option with a number, only the specified warning is suppressed. You can specify the option **--no-warnings=***number* multiple times.

Example

To suppress warnings 135 and 136, enter:

```
ltc --no-warnings=135,136 test.o
```

Related information

Linker option **--warnings-as-errors** (Treat warnings as errors)

Linker option: --optimize (-O)

Menu entry

1. Select **Linker » Optimization**.

2. Select one or more of the following options:

    - Delete unreferenced sections

    - Use a 'first-fit decreasing' algorithm

    - Compress copy table

    - Delete duplicate code

    - Delete duplicate data

Command line syntax

```
--optimize=flag,...
```

```
-Oflags
```

You can set the following flags:

+/-delete-unreferenced-sections	**c/C**	Delete unreferenced sections from the output file
+/-first-fit-decreasing	**l/L**	Use a 'first-fit decreasing' algorithm to locate unrestricted sections in memory
+/-copytable-compression	**t/T**	Emit smart restrictions to reduce copy table size
+/-delete-duplicate-code	**x/X**	Delete duplicate code sections from the output file
+/-delete-duplicate-data	**y/Y**	Delete duplicate constant data from the output file

Use the following options for predefined sets of flags:

--optimize=0	**-O0**	No optimization Alias for **-OCLTXY**
--optimize=1	**-O1**	Default optimization Alias for **-OcLtxy**
--optimize=2	**-O2**	All optimizations Alias for **-Ocltxy**

Default: **--optimize=1**

Description

With this option you can control the level of optimization.

Related information

For details about each optimization see Section 7.7, *Linker Optimizations*.

Linker option: --option-file (-f)

Menu entry

1. Select **Linker » Miscellaneous**.

2. Add the option **--option-file** to the **Additional options** field.

   Be aware that the options in the option file are added to the linker options you have set in the other pages. Only in extraordinary cases you may want to use them in combination.

Command line syntax

--option-file=*file*,...

-f *file*,...

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the linker.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

Option files can also be generated on the fly, for example by the make utility. You can specify the option **--option-file** multiple times.

Format of an option file

• Multiple arguments on one line in the option file are allowed.

• To include whitespace in an argument, surround the argument with single or double quotes.

• If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

```
"This has a single quote ' embedded"

'This has a double quote " embedded'

'This has a double quote " and a single quote '"' embedded"
```

• When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \
line"

        -> "This is a continuation line"
```

• It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
--map-file=my.map              (generate a map file)
test.o                         (input file)
--library-directory=c:\mylibs  (additional search path for system libraries)
```

Specify the option file to the linker:

```
ltc --option-file=myoptions
```

This is equivalent to the following command line:

```
ltc --map-file=my.map test.o --library-directory=c:\mylibs
```

Related information

-

Linker option: --output (-o)

Menu entry

1. Select **Linker » Output Format**.

2. Enable one or more output formats.

   For some output formats you can specify a number of suboptions.

   Eclipse always uses the project name as the basename for the output file.

Command line syntax

--output=[*filename*][**:**format[**:**addr_size][**,**space_name]]...

-o[*filename*][**:**format[**:**addr_size][**,**space_name]]...

You can specify the following formats:

ELF	ELF/DWARF
IHEX	Intel Hex
SREC	Motorola S-records

Description

By default, the linker generates an output file in ELF/DWARF format, with the name `task1.elf`.

With this option you can specify an alternative *filename*, and an alternative output *format*. The default output format is the format of the first input file.

You can use the **--output** option multiple times. This is useful to generate multiple output formats. With the first occurrence of the **--output** option you specify the basename (the filename without extension), which is used for subsequent **--output** options with no filename specified. If you do not specify a filename, or you do not specify the **--output** option at all, the linker uses the default basename `task`*n*.

IHEX and SREC formats

If you specify the Intel Hex format or the Motorola S-records format, you can use the argument *addr_size* to specify the size of addresses in bytes (record length). For Intel Hex you can use the values: 1, 2, and 4 (default). For Motorola S-records you can specify: 2 (S1 records), 3 (S2 records, default) or 4 bytes (S3 records). Note that if you make the *addr_size* too small, the linker might give a fatal object writer error indicating an address overflow.

With the argument *space_name* you can specify the name of the address space. The name of the output file will be filename with the extension `.hex` or `.sre` and contains the code and data allocated in the specified space. If they exist, any other address spaces are also emitted whereas their output files are named *filename_spacename* with the extension `.hex` or `.sre`.

If you do not specify *space_name*, or you specify a non-existing space, the default address space is filled in.

Use option **--chip-output** (**-c**) to create Intel Hex or Motorola S-record output files for each chip defined in the LSL file (suitable for loading into a PROM-programmer).

Example

To create the output file `myfile.hex` of the address space named `linear`, enter:

```
ltc test.o --output=myfile.hex:IHEX:2,linear
```

If they exist, any other address spaces are emitted as well and are named `myfile_`*spacename*`.hex`.

Related information

Linker option **--chip-output** (Generate an output file for each chip)

Linker option **--hex-format** (Specify Hex file format settings)

Linker option: --print-mangled-symbols (-P)

Menu entry

-

Command line syntax

`--print-mangled-symbols`

`-P`

Description

C++ compilers generate unreadable symbol names. These symbols cannot easily be related to your C++ source file anymore. Therefore the linker will by default decode these symbols conform the IA64 ABI when printed to stdout. With this option you can override this default setting and print the mangled names instead.

Related information

-

Linker option: --software-partitioning-info

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

2. Enable the option **Check software partitioning**.

Eclipse also calls the integrity check tool.

Command line syntax

```
--software-partitioning-info
```

Description

With this option the linker adds information about section access rights to the output file in
`.debug_tasking_verif.rights` sections. This information can be used by the integrity check utility.

Related information

Section 8.10, *Integrity Check Utility*

Section 8.6.2, *HLL Dump Output Format*

Linker option: --strip-debug (-S)

Menu entry

1. Select **Linker » Miscellaneous**.

2. Enable the option **Strip symbolic debug information**.

Command line syntax

`--strip-debug`

`-s`

Description

With this option you specify not to include symbolic debug information in the resulting output file.

Related information

-

Linker option: --user-provided-initialization-code (-i)

Menu entry

1. Select **Linker » Miscellaneous**.

2. Enable the option **Do not use standard copy table for initialization**.

Command line syntax

```
--user-provided-initialization-code
```

```
-i
```

Description

It is possible to use your own initialization code, for example, to save ROM space. With this option you tell the linker *not* to generate a copy table for initialize/clear sections. Use linker labels in your source code to access the positions of the sections when located.

If the linker detects references to the TASKING initialization code, an error is emitted: it is either the TASKING initialization routine or your own, not both.

Note that the options **--no-rom-copy** and **--non-romable**, may vary independently. The 'copytable-compression' optimization (**--optimize=t**) is automatically disabled when you enable this option.

Related information

Linker option **--no-rom-copy** (Do not generate ROM copy)

Linker option **--non-romable** (Application is not romable)

Linker option **--optimize** (Specify optimization)

Linker option: --verbose (-v)

Menu entry

1. Select **Linker » Miscellaneous**.

2. Enable the option **Show link phases during processing**.

   The verbose output is displayed in the Problems view and the Console view.

Command line syntax

--verbose

-v

Description

With this option you put the linker in *verbose* mode. The linker prints the link phases while it processes the files. The linker prints one entry for each action it executes for a task. When you use this option twice (**-vv**) you put the linker in *extra verbose* mode. In this mode the linker also prints the filenames and it shows which objects are extracted from libraries and it shows verbose information that would normally be hidden when you use the normal verbose mode or when you run without verbose. With this option you can monitor the current status of the linker.

Related information

-

Linker option: --version (-V)

Menu entry

-

Command line syntax

`--version`

`-V`

Description

Display version information. The linker ignores all other options or input files.

Example

```
ltc --version
```

The linker does not link any files but displays the following version information:

```
TASKING VX-toolset for TriCore: object linker    vx.yrz Build nnn
Copyright 2002-year Altium BV                    Serial# 00000000
```

Related information

-

Linker option: --warnings-as-errors

Menu entry

1. Select **Global Options**.

2. Enable the option **Treat warnings as errors**.

Command line syntax

```
--warnings-as-errors[=number,...]
```

Description

When the linker detects an error or warning, it tries to continue the link process and reports other errors and warnings. When you use this option without arguments, you tell the linker to treat all warnings as errors. This means that the exit status of the linker will be non-zero after the detection of one or more linker warnings. As a consequence, the linker will not produce any output files.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

Linker option **--no-warnings** (Suppress some or all warnings)

Linker option: --whole-archive

Menu entry

1. Select **Linker » Miscellaneous**.

2. Add the option **--whole-archive** to the **Additional options** field.

Command line syntax

--whole-archive=*file*

Description

This option tells the linker to directly load all object modules in a library, as if they were placed on the command line. This is different from libraries specified as input files or with the **-l** option, which are only used to resolve references in object files that were loaded earlier.

Example

Suppose the library `myarchive.a` contains the objects `my1.o`, `my2.o` and `my3.o`. Specifying

```
ltc --whole-archive=myarchive.a
```

is the same as specifying

```
ltc my1.o my3.o my3.o
```

Related information

Linker option **--library** (Link system library)

10.6. Control Program Options

The control program **cctc** facilitates the invocation of the various components of the TriCore toolset from a single command line.

Options in Eclipse versus options on the command line

Eclipse invokes the compiler, assembler and linker via the control program. Therefore, it uses the syntax of the control program to pass options and files to the tools. The control program processes command line options either by itself, or, when the option is unknown to the control program, it looks whether it can pass the option to one of the other tools. However, for directly passing an option to the C++ compiler, C compiler, assembler or linker, it is recommended to use the control program options **--pass-c++**, **--pass-c**, **--pass-assembler**, **--pass-linker**.

See the previous sections for details on the options of the tools.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+***longflag*. To switch a flag off, use an uppercase letter or a **-***longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
cctc -Wc-Oac test.c
cctc --pass-c=--optimize=+coalesce,+cse test.c
```

When you do not specify an option, a default value may become active.

Control program option: --address-size

Menu entry

1. Select **Linker » Output Format**.

2. Enable the option **Generate Intel Hex format file** and/or **Generate S-records file**.

3. Specify the **Size of addresses**.

   Eclipse always uses the project name as the basename for the output file.

Command line syntax

--address-size=*addr_size*

Description

If you specify IHEX or SREC with the control option **--format**, you can additionally specify the record length to be emitted in the output files.

With this option you can specify the size of the addresses in bytes (record length). For Intel Hex you can use the values **1**, **2** or **4** bytes (default). For Motorola-S you can specify: **2** (S1 records), **3** (S2 records) or **4** bytes (S3 records, default).

If you do not specify *addr_size*, the default address size is generated.

Example

To create the SREC file `test.sre` with S1 records, type:

```
cctc --format=SREC --address-size=2 test.c
```

Related information

Control program option **--format** (Set linker output format)

Control program option **--output** (Output file)

Control program option: --c++11

Menu entry

1. Select **C/C++ Compiler » Language**.

2. From the **Comply to C++ standard** list, select **ISO C++11**.

Command line syntax

`--c++11`

Description

Enable the C++ extensions that are defined by the C++11 standard (ISO/IEC 14882:2011). This option also enables exceptions.

Related information

Control program option **--exceptions** (Enable exceptions)

Control program option: --c++14

Menu entry

1. Select **C/C++ Compiler » Language**.

2. From the **Comply to C++ standard** list, select **ISO C++14**.

Command line syntax

`--c++14`

Description

Enable the C++ extensions that are defined by the C++14 standard (ISO/IEC 14882:2014). This option also enables exceptions.

Related information

Control program option **--exceptions** (Enable exceptions)

Control program option: --case-insensitive

Menu entry

1. Select **Assembler » Symbols**.

2. Enable the option **Case insensitive identifiers**.

Command line syntax

`--case-insensitive`

Default: case sensitive

Description

With this option you tell the assembler not to distinguish between uppercase and lowercase characters. By default the assembler considers uppercase and lowercase characters as different characters.

Assembly source files that are generated by the compiler must always be assembled case sensitive. When you are writing your own assembly code, you may want to specify the case insensitive mode.

Example

When assembling case insensitive, the label `LabelName` is the same label as `labelname`.

```
cctc --case-insensitive test.src
```

Related information

Assembler option **--case-insensitive**

Assembler control **$CASE**

Control program option: --check

Menu entry

-

Command line syntax

`--check`

Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application because the code will not actually be compiled.

The compiler/assembler reports any warnings and/or errors.

This option is available on the command line only.

Related information

C compiler option **--check** (Check syntax)

Assembler option **--check** (Check syntax)

Control program option: --command-file

Menu entry

1. Select **Linker » Patching**.

2. Enable the option **Patch object and library files supplied to the linker**.

3. Specify an **ELF patch command file**.

Command line syntax

--command-file=*file*

Description

Call the **elfpatch** utility before linking and execute the commands from the specified ELF patch command file (EPF). Such a file contains one or more `rename_section` commands and/or `rename_section_by_symbol` commands.

Example

```
cctc test.c -Ctc27x --command-file=commands.epf
```

Related information

Section 8.7, *ELF Patch Utility*

ELF patch utility option **--command-file**

Control program option: --control-flow-info

Menu entry

1.   Select **C/C++ Compiler » Debugging**.

2.   Enable the option **Generate control flow information**.

Command line syntax

```
--control-flow-info
```

Description

Control flow information

With this option the compiler adds control flow information to the output file. The compiler generates a `.debug_control_flow` section which describes the basic blocks and their relations. This information can be used for code coverage analysis on optimized code.

Example

```
cctc --control-flow-info test.c
```

Related information

Section 8.6.2, *HLL Dump Output Format*

Control program option **--debug-info** (Debug information)

Control program option: --core

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor Selection** list, select a processor or select **User defined TriCore ...**.

Command line syntax

--core=*core*

You can specify the following *core* arguments:

tc1.3	TriCore 1.3 architecture
tc1.3.1	TriCore 1.3.1 architecture, TriCore 1.3.1 instructions may be generated
tc1.6	TriCore 1.6 architecture, TriCore 1.6 instructions may be generated
tc1.6.x	TriCore 1.6.x architecture, TriCore 1.6.x instructions may be generated
tc1.6.2	TriCore 1.6.2 architecture, TriCore 1.6.2 instructions may be generated

Default: **tc1.3**

Description

With this option you specify the core architecture for a target processor for which you create your application. If you use Eclipse or the control program, the TriCore toolset derives the core from the processor you selected. When the default core (**tc1.3**) is used, the macro __CORE_TC13__ is defined in the C source file.

With **--core=tc1.3.1**, the compiler can generate TriCore 1.3.1 instructions in the assembly file. The macro __CORE_TC131__ is defined in the C source file.

With **--core=tc1.6**, the compiler can generate TriCore 1.6 instructions in the assembly file. The macro __CORE_TC16__ is defined in the C source file.

With **--core=tc1.6.x**, the compiler can generate TriCore 1.6.x instructions in the assembly file. The macro __CORE_TC16X__ is defined in the C source file.

With **--core=tc1.6.2**, the compiler can generate TriCore 1.6.2 instructions in the assembly file. The macro __CORE_TC162__ is defined in the C source file.

If you select a valid target processor (command line option **--cpu** (**-C**)), the core is automatically set, based on the chosen target processor.

Example

To allow the use of TriCore 1.3.1 instructions in the assembly code, enter:

```
cctc --core=tc1.3.1 test.c
```

Related information

Control program option **--cpu** (Select processor)

Control program option **--mmu-present** (MMU present)

Control program option **--fp-model=+soft** (Disable hardware floating-point instructions)

Control program option: --cpu (-C)

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor Selection** list, select a processor.

Command line syntax

--cpu=*id* | *name* | *cpu*

-C*id* | *name* | *cpu*

Description

With this option you define the target processor for which you create your application. You can specify a full processor *name*, like TC1796B, or a base *CPU* name, like tc1796b or its unique *id*, like tc1796b.

Based on this option the compiler can include the special function register file reg*cpu*.sfr, and the assembler can include the file reg*cpu*.def, if you specify option **--tasking-sfr**.

Based on the target processor it is automatically detected whether a MMU or FPU-unit is present and whether the architecture is a TriCore1.6. This means you do not have to specify the options **--mmu-present**, **--fp-model=+soft** and **--core=tc1.6** explicitly when one of the supported derivatives is selected.

The standard list of supported processors is defined in the file processors.xml. This file defines for each processor its full name (for example, TC1796B), its ID, the base CPU name (for example, tc1796b), the core settings (for example, tc1.3), the MMU and FPU settings and the list of silicon bugs for that processor. To show a list of all supported processors you can use option **--cpu-list**.

The control program reads the file processors.xml. The lookup sequence for names specified to this option is as follows:

1. match with the 'id' attribute in processors.xml (case insensitive, for example tc1796b)

2. if none matched, match with the 'name' attribute in processors.xml (case insensitive, for example TC1796B)

3. if still none matched, match any of the base CPU names (the 'cpu' attribute in processors.xml, for example tc1796b). If multiple processors exist with the same base CPU, a warning will be issued and the first one is selected.

4. if still none matched, the control program issues a fatal error.

The preferred use of the option **--cpu**, is to specify an ID because that is always a unique name. For example, **--cpu=tc1796b**. The control program will lookup this processor name in the file processors.xml. The control program passes the options to the underlaying tools. For example, **-D__CPU__=tc1796b -D__CPU_TC1796B__ --core=tc1.3** to the C compiler, or **-dtc1796b.lsl -D__CPU__=tc1796b -D__PROC_TC1796B__** to the linker.

Example

To generate the file `test.elf` for the TC1796B processor, enter:

```
cctc --cpu=tc1796b -v -t test.c
```

The control program will call the tools as follows:

```
ctc   -D__CPU__=tc1796b -D__CPU_TC1796B__  --core=tc1.3
      --fp-model=+float -o test.src test.c
astc  -D__CPU__=tc1796b -D__CPU_TC1796B__  --core=tc1.3
      -o test.o test.src
lktc  -o test.elf -dtc1796b.lsl -dextmem.lsl -D__CPU__=tc1796b
      -D__PROC_TC1796B__ --map-file test.o -lcs_fpu -lfp_fpu
      -lrt -Linstall-dir/ctc/lib/tc13
```

Related information

Control program option **--cpu-list** (Show list of processors)

Control program option **--processors** (Read additional processor definitions)

Control program option **--tasking-sfr** (Include SFR file)

Section 1.3.2, *Accessing Hardware from C*

Control program option: --cpu-list

Menu entry

-

Command line syntax

--cpu-list[=*pattern*]

Description

With this option the control program shows a list of supported processors as defined in the file `processors.xml`. This can be useful when you want to select a processor name or id for the **--cpu** option.

The *pattern* works similar to the UNIX **grep** utility. You can use it to limit the output list.

Example

To show a list of all processors, enter:

```
cctc --cpu-list
```

To show all processors of the tc1.6.x core, enter:

```
cctc --cpu-list=tc1.6.x

--- ~/ctc/etc/processors.xml ---
    id          name                      CPU         core
    userdef16x  User defined TriCore 1.6.x  userdef16x  tc1.6.x
    tc21x       TC21x                      tc21x       tc1.6.x
    tc22x       TC22x                      tc22x       tc1.6.x
    tc23x       TC23x                      tc23x       tc1.6.x
    tc23x_adas  TC23x_ADAS                 tc23x_adas  tc1.6.x
    tc26x       TC26x                      tc26x       tc1.6.x
    tc26xb      TC26xB                     tc26xb      tc1.6.x
    tc27x       TC27x                      tc27x       tc1.6.x
    tc27xb      TC27xB                     tc27xb      tc1.6.x
    tc27xc      TC27xC                     tc27xc      tc1.6.x
    tc27xd      TC27xD                     tc27xd      tc1.6.x
    tc29x       TC29x                      tc29x       tc1.6.x
    tc29xb      TC29xB                     tc29xb      tc1.6.x
```

Related information

Control program option **--cpu** (Select processor)

Control program option: --create (-c)

Menu entry

-

Command line syntax

--create[**=***stage*]

-c[*stage*]

You can specify the following stages:

intermediate-c	**c**	Stop after C++ files are compiled to intermediate C files (`.ic`)
relocatable	**l**	Stop after the files are linked to a linker object file (`.out`)
mil	**m**	Stop after C++ files or C files are compiled to MIL (`.mil`)
object	**o**	Stop after the files are assembled to objects (`.o`)
assembly	**s**	Stop after C++ files or C files are compiled to assembly (`.src`)

Default (without flags): **--create=object**

Description

Normally the control program generates an absolute object file of the specified output format from the file you supplied as input. With this option you tell the control program to stop after a certain number of phases.

Example

To generate the object file `test.o`:

```
cctc --create test.c
```

The control program stops after the file is assembled. It does not link nor locate the generated output.

Related information

Control program option **--link-only** (Link only, no locating)

Control program option: --data-reference-modification-file

Menu entry

1. Select **Linker » Patching**.

2. Enable the option **Patch object and library files supplied to the linker**.

3. Specify a **Data reference modification file**.

4. From the **Add info section to patched object files** list, select **Full**, **Compact**, **Executed commands** or **None**.

Command line syntax

```
--data-reference-modification-file=file
```

Description

Call the **elfpatch** utility before linking and execute the commands from the specified ELF patch data reference modification file (DRM). Such a file contains one or more commands to specify a replacement for any access to a specified global variable.

By default the contents of this file is placed in a section called `.VarEd_Info`, unless you specify option **--vared-info-section=no-section**.

Example

To read data reference modifications from file `myproject.drm`, enter:

```
cctc test.c –Ctc27x --data-reference-modification-file=myproject.drm
```

Related information

Section 8.7, *ELF Patch Utility*

Control program option **--vared-info-section**

Control program option: --debug-info (-g)

Menu entry

1. Select **C/C++ Compiler » Debugging**.

2. To **generate symbolic debug information**, select **Default**, **Small set** or **Full**.
   To disable the generation of debug information, select **None**.

Command line syntax

--debug-info[**=**_suboption_]

-g[_suboption_]

You can set the following suboptions:

small	**1** | **c**	Emit small set of debug information.
default	**2** | **d**	Emit default symbolic debug information.
all	**3** | **a**	Emit full symbolic debug information.

Default: **--debug-info** (same as **--debug-info=default**)

Description

With this option you tell the control program to include debug information in the generated object file.

The control program passes the option **-g**_suboption_ to the C compiler and calls the assembler with **-gsl**.

Related information

C compiler option **--debug-info** (Generate symbolic debug information)

Assembler option **--debug-info** (Generate symbolic debug information)

Control program option: --define (-D)

Menu entry

1.   Select **C/C++ Compiler » Preprocessing** and/or **Assembler » Preprocessing**.

     The Defined symbols box right-below shows the symbols that are currently defined.

2.   To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3.   Type the symbol definition (for example, demo=1)

Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

--define=*macro_name*[**=***macro_definition*]

-D*macro_name*[**=***macro_definition*]

Description

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the compiler with the option **--option-file** (**-f**) *file*.

Defining macros with this option (instead of in the C source) is, for example, useful to compile conditional C source as shown in the example below.

The control program passes the option **--define** (**-D**) to the compiler and the assembler.

Make sure you do not use a reserved keyword as a macro name, as this can lead to unexpected results.

Example

Consider the following C program with conditional code to compile a demo program and a real program:

```
void main( void )
{
#if DEMO
    demo_func();   /* compile for the demo program */
#else
    real_func();   /* compile for the real program */
```

```
#endif
}
```

You can now use a macro definition to set the DEMO flag:

```
cctc --define=DEMO test.c
cctc --define=DEMO=1 test.c
```

Note that both invocations have the same effect.

The next example shows how to define a macro with arguments. Note that the macro name and definition are placed between double quotes because otherwise the spaces would indicate a new option.

```
cctc --define="MAX(A,B)=((A) > (B) ? (A) : (B))" test.c
```

Related information

Control program option **--undefine** (Remove preprocessor macro)

Control program option **--option-file** (Specify an option file)

Control program option: --dep-file

Menu entry

-

Command line syntax

--dep-file[**=***file*]

Description

With this option you tell the compiler to generate dependency lines that can be used in a Makefile. In contrast to the option **--preprocess=+make**, the dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension .d (one for every input file). When you specify a filename, all dependencies will be combined in the specified file.

Example

```
cctc --dep-file=test.dep -t test.c
```

The compiler compiles the file test.c, which results in the output file test.src, and generates dependency lines in the file test.dep.

Related information

Control program option **--preprocess=+make** (Generate dependencies for make)

Control program option: --diag

Menu entry

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

    The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

    A popup menu appears.

3. Select **Detailed Diagnostics Info**.

    A dialog box appears with additional information.

Command line syntax

--diag=[*format*:]{**all** | *msg*[-*msg*],...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas, or you can specify a range.

Example

To display an explanation of message number 103, enter:

```
cctc --diag=103
```

This results in message 103 with explanation.

To write an explanation of all errors and warnings in HTML format to file `ccerrors.html`, use redirection and enter:

```
cctc --diag=html:all > ccerrors.html
```

Related information

Section 4.8, *C Compiler Error Messages*

Control program option: --dry-run (-n)

Menu entry

-

Command line syntax

`--dry-run`

`-n`

Description

With this option you put the control program in verbose mode. The control program prints the invocations of the tools it would use to process the files without actually performing the steps.

Related information

Control program option **--verbose** (Verbose output)

Control program option: --error-file

Menu entry

-

Command line syntax

`--error-file`

Description

With this option the control program tells the compiler, assembler and linker to redirect error messages to a file.

The error file will be named after the output file with extension `.err` (for compiler) or `.ers` (for assembler). For the linker, the error file is `ltc.elk`.

Example

To write errors to error files instead of stderr, enter:

```
cctc --error-file test.c
```

Related information

Control Program option **--warnings-as-errors** (Treat warnings as errors)

Control program option: --exceptions

Menu entry

1.   Select **C/C++ Compiler » Language**.

2.   Enable or disable the option **Support for C++ exception handling**.

Command line syntax

`--exceptions`

`--no-exceptions`

Default: **`--no-exceptions`**

Default in C++11/C++14 mode: **`--exceptions`**

Description

With this option you enable or disable support for exception handling in the C++ compiler.

The macro __EXCEPTIONS is defined when exception handling support is enabled.

This feature is implicitly enabled in C++11 mode and C++14 mode.

Related information

Control program option **--c++11** (C++11 language extensions)

Control program option **--c++14** (C++14 language extensions)

Control program option: --force-c

Menu entry

-

Command line syntax

```
--force-c
```

Description

With this option you tell the control program to treat all `.cc` files as C files instead of C++ files. This means that the control program does not call the C++ compiler and forces the linker to link C libraries.

Related information

Control program option **--force-c++** (Force C++ compilation and linking)

Control program option: --force-c++

Menu entry

Eclipse always uses this option for a C++ project.

Command line syntax

```
--force-c++
```

Description

With this option you tell the control program to treat all `.c` files as C++ files instead of C files. This means that the control program calls the C++ compiler prior to the C compiler and forces the linker to link C++ libraries.

Related information

Control program option **--force-c** (Treat C++ files as C files)

Control program option: --force-munch

Menu entry

Eclipse always uses this option for a C++ project.

Command line syntax

`--force-munch`

Description

With this option you force the control program to activate the muncher in the pre-locate phase.

Related information

-

Control program option: --format

Menu entry

1. Select **Linker » Output Format**.

2. Enable the option **Generate Intel Hex format file** and/or **Generate S-records file**.

3. Optionally, specify the **Size of addresses**.

   Eclipse always uses the project name as the basename for the output file.

Command line syntax

--format=`format`

You can specify the following formats:

ELF	ELF/DWARF
IHEX	Intel Hex
SREC	Motorola S-records

Description

With this option you specify the output format for the resulting (absolute) object file. The default output format is ELF/DWARF, which can directly be used by the debugger.

If you choose IHEX or SREC, you can additionally specify the address size of the chosen format (option **--address-size**).

Example

To generate a Motorola S-record output file:

```
cctc --format=SREC test1.c test2.c --output=test.sre
```

Related information

Control program option **--address-size** (Set address size for linker IHEX/SREC files)

Control program option **--output** (Output file)

Linker option **--chip-output** (Generate an output file for each chip)

Control program option: --fp-model

Menu entry

1. Select **C/C++ Compiler » Floating-Point**.

2. Make a selection from the **Floating-point model** list.

3. If you selected **Custom**, enable one or more individual options.

Command line syntax

--fp-model=*flags*

You can set the following flags:

+/-contract	**c/C**	allow expression contraction
+/-float	**f/F**	treat 'double' as 'float'
+/-fastlib	**l/L**	allow less precise library functions
+/-nonan	**n/N**	allow optimizations to ignore NaN/Inf
+/-rewrite	**r/R**	allow expression rewriting
+/-soft	**s/S**	use software floating-point library
+/-trap	**t/T**	support trapping on exceptions
+/-negzero	**z/Z**	ignore sign of -0.0

0	alias for **--fp-model=CFLNRStZ** (strict)
1	alias for **--fp-model=cFLNRSTZ** (precise)
2	alias for **--fp-model=cFlnrSTz** (fast double)
3	alias for **--fp-model=cflnrSTz** (fast single)

Default: **--fp-model=cflnrSTz**

Description

With this option you select the floating-point execution model.

With **--fp-model=+contract** you allow the compiler to contract multiple float operations into a single operation, with different rounding results. A possible example is fused multiply-add.

With **--fp-model=+float** you tell the compiler to treat variables and constants of type double as float. Because the float type takes less space, execution speed increases and code size decreases, both at the cost of less precision. The control program automatically selects the correct libraries.

With **--fp-model=+fastlib** you allow the compiler to select faster but less accurate library functions for certain floating-point operations. With **--fp-model=-fastlib** more precise library functions are used and the compiler defines the macro __PRECISE_LIB_FP__, which is used in math.h.

With **--fp-model=+nonan** you allow the compiler to ignore NaN or Inf input values. An example is to replace multiply by zero with zero.

With **--fp-model=+rewrite** you allow the compiler to rewrite expressions by reassociating. This might result in rounding differences and possibly different exceptions. An example is to rewrite (a*c)+(b*c) as (a+b)*c.

With **--fp-model=+soft** no hardware floating-point instructions are generated, only calls to the software floating-point library.

By default, the compiler assumes an FPU is present and the macro __FPU__ is defined in the C source file. With this option the compiler does not generate single precision floating-point instructions in the assembly file. When you select this option, the macro __FPU__ is not defined in the C source file and the compiler generates the assembler control $NO_FPU in the assembly source.

With **--fp-model=+trap** operations trap on floating-point exceptions. By default the control program uses the non-trapping versions of the floating-point library (libfp.a or libfp_fpu.a). With this option you tell the control program to use the trapping versions of the floating-point library (libfpt.a or libfpt_fpu.a).

If you use the trapping floating-point library, exceptional floating-point cases are intercepted and can be handled separately by an application defined exception handler. Using this library decreases the execution speed of your application.

With **--fp-model=+negzero** you allow the compiler to ignore the sign of -0.0 values. An example is to replace (a-a) by zero.

Related information

Pragmas STDC FP_CONTRACT, fp_negzero, fp_nonan and fp_rewrite in Section 1.8, *Pragmas to Control the Compiler*.

C compiler option **--eabi=+float** (allow treat 'double' as 'float')

Control program option: --global-type-checking

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

2. Enable the option **Perform global type checking on C code**.

Command line syntax

```
--global-type-checking
```

Description

The C compiler already performs type checking within each module. Use this option when you want the linker to perform type checking between modules. The control program passes this option to both the C compiler and the linker.

Related information

-

Control program option: --help (-?)

Menu entry

-

Command line syntax

--help[**=**item]

-?

You can specify the following argument:

options   Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

The following invocations all display a list of the available command line options:

```
cctc -?
cctc --help
cctc
```

To see a detailed description of the available options, enter:

```
cctc --help=options
```

Related information

-

Control program option: --include-directory (-I)

Menu entry

1.   Select **C/C++ Compiler » Include Paths**.

     The Include paths box shows the directories that are added to the search path for include files.

2.   To define a new directory for the search path, click on the **Add** button in the **Include paths** box.

3.   Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

--include-directory=*path*,...

-I*path*,...

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory.

The control program passes this option to the compiler and the assembler.

Example

Suppose that the C source file `test.c` contains the following lines:

```
#include <stdio.h>
#include "myinc.h"
```

You can call the control program as follows:

```
cctc --include-directory=myinclude test.c
```

First the compiler looks for the file `stdio.h` in the directory `myinclude` relative to the current directory. If it was not found, the compiler searches in the environment variable and then in the default include directory.

The compiler now looks for the file `myinc.h` in the directory where `test.c` is located. If the file is not there the compiler searches in the directory `myinclude`. If it was still not found, the compiler searches in the environment variable and then in the default include directory.

Related information

C compiler option **--include-directory** (Add directory to include file search path)

C compiler option **--include-file** (Include file at the start of a compilation)

Control program option: --include-file (-H)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

    The Pre-include files box shows the files that are currently included before the compilation starts.

2. To define a new file, click on the **Add** button in the **Pre-include files** box.

3. Type the full path and file name or select a file.

Use the **Edit** and **Delete** button to change a file name or to remove a file from the list.

Command line syntax

--include-file=*file***,...**

-H*file***,...**

Description

With this option you include one or more extra files at the beginning of each C source file, before other includes. This is the same as specifying `#include "file"` at the beginning of *each* of your C sources.

Example

```
cctc --include-file=stdio.h test1.c test2.c
```

The file `stdio.h` is included at the beginning of both `test1.c` and `test2.c`.

Related information

C compiler option **--include-directory** (Add directory to include file search path)

C compiler option **--include-file** (Include file at the start of a compilation)

Control program option: --instantiate

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Select an instantiation mode in the **Instantiation mode of external template entities** box.

Command line syntax

--instantiate=*mode*

You can specify the following modes:

> **used**
>
> **all**
>
> **local**

Default: **--instantiate=used**

Description

Control instantiation of external template entities. External template entities are external (that is, non-inline and non-static) template functions and template static data members. The instantiation mode determines the template entities for which code should be generated based on the template definition. Normally, when a file is compiled, template entities are instantiated wherever they are used (the linker will discard duplicate definitions). The overall instantiation mode can, however, be changed with this option. You can specify the following modes:

used	Instantiate those template entities that were used in the compilation. This will include all static data members for which there are template definitions. This is the default.
all	Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members will be instantiated whether or not they were used. Non-member template functions will be instantiated even if the only reference was a declaration.
local	Similar to **--instantiate=used** except that the functions are given internal linkage. This is intended to provide a very simple mechanism for those getting started with templates. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly (barring problems due to multiple copies of local static variables). However, one may end up with many copies of the instantiated functions, so this is not suitable for production use.

You cannot use **--instantiate=local** in conjunction with automatic template instantiation.

Related information

Section 2.4, *Template Instantiation*

Control program option: --integer-enumeration

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Always use 32-bit integers for enumeration**.

Command line syntax

`--integer-enumeration`

Description

Normally the compiler treats enumerated types as the smallest data type possible (`char` or `short` instead of `int`). This reduces code size. With this option the compiler always treats enum-types as `int` as defined in the ISO C99 standard.

Related information

Section 1.1, *Data Types*

Control program option: --io-streams

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Support for C++ I/O streams**.

Command line syntax

```
--io-streams
```

Description

As I/O streams require substantial resources they are disabled by default. Use this option to enable I/O streams support in the C++ library.

This option also enables exception handling.

Related information

-

Control program option: --iso

Menu entry

1. Select **C/C++ Compiler » Language**.

2. From the **Comply to C standard** list, select **ISO C99** or **ISO C90**.

Command line syntax

--iso={**90**|**99**}

Default: **--iso=99**

Description

With this option you select the ISO C standard. C90 is also referred to as the "ANSI C standard". C99 refers to the newer ISO/IEC 9899:1999 (E) standard. C99 is the default.

Independent of the chosen ISO standard, the control program always links libraries with C99 support.

Example

To select the ISO C90 standard on the command line:

```
cctc --iso=90 test.c
```

Related information

C compiler option **--iso** (ISO C standard)

Control program option: --keep-output-files (-k)

Menu entry

Eclipse *always* removes generated output files when an error occurs.

Command line syntax

`--keep-output-files`

`-k`

Description

If an error occurs during the compilation, assembling or linking process, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the control program removes generated output files when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated files. For example when you know that a particular error does not result in a corrupt file, or when you want to inspect the output file, or send it to Altium support.

The control program passes this option to the compiler, assembler and linker.

Example

```
cctc --keep-output-files test.c
```

When an error occurs during compiling, assembling or linking, the erroneous generated output files will not be removed.

Related information

C compiler option **--keep-output-files**

Assembler option **--keep-output-files**

Linker option **--keep-output-files**

Control program option: --keep-temporary-files (-t)

Menu entry

1. Select **Global Options**.

2. Enable the option **Keep temporary files**.

Command line syntax

```
--keep-temporary-files
```

```
-t
```

Description

By default, the control program removes intermediate files like the `.src` file (result of the compiler phase) and the `.o` file (result of the assembler phase).

With this option you tell the control program to keep temporary files it generates during the creation of the absolute object file.

Example

```
cctc --keep-temporary-files test.c
```

The control program keeps all intermediate files it generates while creating the absolute object file `test.elf`.

Related information

-

Control program option: --library (-l)

Menu entry

1. Select **Linker » Libraries**.

    The Libraries box shows the list of libraries that are linked with the project.

2. To add a library, click on the **Add** button in the **Libraries** box.

3. Type or select a library (including its path).

4. Optionally, disable the option **Link default libraries**.

> Use the **Edit** and **Delete** button to change a library name or to remove a library from the list.

Command line syntax

`--library=``name`

`-l``name`

Description

With this option you tell the linker via the control program to use system library lib*name*.a, where *name* is a string. The linker first searches for system libraries in any directories specified with **--library-directory**, then in the directories specified with the environment variables LIBTC1V1_3 / LIBTC1V1_3_1 / LIBTC1V1_6 / LIBTC1V1_6_X / LIBTC1V1_6_2, unless you used the option **--ignore-default-library-path**.

Example

To search in the system library libc.a (C library):

```
cctc test.o mylib.a --library=c
```

The linker links the file test.o and first looks in library mylib.a (in the current directory only), then in the system library libc.a to resolve unresolved symbols.

Related information

Control program option **--no-default-libraries** (Do not link default libraries)

Control program option **--library-directory** (Additional search path for system libraries)

Section 7.3, *Linking with Libraries*

Control program option: --library-directory (-L) / --ignore-default-library-path

Menu entry

1.   Select **Linker » Libraries**.

     The Library search path box shows the directories that are added to the search path for library files.

2.   To define a new directory for the search path, click on the **Add** button in the **Library search path** box.

3.   Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

```
--library-directory=path,...
-Lpath,...

--ignore-default-library-path
-L
```

Description

With this option you can specify the path(s) where your system libraries, specified with the option **--library** (**-l**), are located. If you want to specify multiple paths, use the option **--library-directory** for each separate path.

The default path is
`$(PRODDIR)\lib\[tc13][tc131][tc13_mmu][tc16][tc16_mmu][tc16x][tc162]`.

If you specify only **-L** (without a pathname) or the long option **--ignore-default-library-path**, the linker will not search the default path and also not in the paths specified in the environment variables `LIBTC1V1_3` / `LIBTC1V1_3_1` / `LIBTC1V1_6` / `LIBTC1V1_6_X` / `LIBTC1V1_6_2`. So, the linker ignores steps 2 and 3 as listed below.

The priority order in which the linker searches for system libraries specified with the option **--library** (**-l**) is:

1. The path that is specified with the option **--library-directory**.

2. The path that is specified in the environment variables `LIBTC1V1_3` / `LIBTC1V1_3_1` / `LIBTC1V1_3_1` / `LIBTC1V1_6` / `LIBTC1V1_6_X` / `LIBTC1V1_6_2`.

3. The default directory
   `$(PRODDIR)\lib\[tc13][tc131][tc13_mmu][tc16][tc16_mmu][tc16x][tc162]`.

Example

Suppose you call the control program as follows:

```
cctc test.c --library-directory=c:\mylibs --library=c
```

First the linker looks in the directory `c:\mylibs` for library `libc.a` (this option). If it does not find the requested libraries, it looks in the directory that is set with the environment variables `LIBTC1V1_3` / `LIBTC1V1_3_1` / `LIBTC1V1_6` / `LIBTC1V1_6_X` / `LIBTC1V1_6_2`. Then the linker looks in the default directory
`$(PRODDIR)\lib\[tc13][tc131][tc13_mmu][tc16][tc16_mmu][tc16x][tc162]` for libraries.

Related information

Control program option **--library** (Link system library)

Section 7.3.1, *How the Linker Searches Libraries*

Control program option: --link-only

Menu entry

-

Command line syntax

`--link-only`

Description

With this option you suppress the locating phase. The linker stops after linking and informs you about unresolved references.

Related information

Control program option **--create=relocatable** (**-cl**) (Stop after linking)

Linker option **--link-only** (Link only, no locating)

Control program option: --list-files

Menu entry

-

Command line syntax

--list-files[**=***file*]

Default: no list files are generated

Description

With this option you tell the assembler via the control program to generate a list file for each specified input file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

With the optional *file* you can specify a name for the list file. This is only possible if you specify only one input file to the control program. If you do not specify a file name, or you specify more than one input file, the control program names the generated list file(s) after the specified input file(s) with extension `.lst`.

Note that object files and library files are not counted as input files.

Related information

Assembler option **--list-file** (Generate list file)

Assembler option **--list-format** (Format list file)

Control program option: --lsl-core

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Multi-core configuration** list, select a TriCore single-core or multi-core.

Command line syntax

```
--lsl-core=core
```

You can specify the following cores:

tc0	TriCore core 0
vtc	Multi-core TriCore

Default: **vtc**

Description

With this option you select the core name as specified in a linker script file, for the current task. The control program passes the correct LSL file to the linker and specifies the correct core.

Example

If you call the control program with the following options:

```
-Ctc27x --lsl-core=tc0
```

the control program calls the linker with the following options:

```
-dtc27x_tc0.lsl --core=mpe:tc0
```

Related information

Linker option **--core** (Specify LSL core)

Control program option: --lsl-file (-d)

Menu entry

An LSL file can be generated when you create your project in Eclipse:

1. From the **File** menu, select **File » New » TASKING TriCore C/C++ Project**.

    The New C/C++ Project wizard appears.

2. Fill in the project settings in each dialog and click **Next >** until the **TriCore Project Settings** appear.

3. Enable the option **Add linker script file to the project** and click **Finish**.

    Eclipse creates your project and the file project.lsl in the project directory.

The LSL file can be specified in the Properties dialog:

1. Select **Linker » Script File**.

2. Specify a LSL file in the **Linker script file (.lsl)** field (default `../${ProjName}.lsl`).

Command line syntax

--lsl-file=`file`,...

-d`file`,...

Description

A linker script file contains vital information about the core for the locating phase of the linker. A linker script file is coded in LSL and contains the following types of information:

*   the architecture definition describes the core's hardware architecture.

*   the memory definition describes the physical memory available in the system.

*   the section layout definition describes how to locate sections in memory.

With this option you specify a linker script file via the control program to the linker. If you do not specify this option, the linker uses a default script file. You can specify the existing file *target*.lsl or the name of a manually written linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

Related information

Section 7.8, *Controlling the Linker with a Script*

Control program option: --make-target

Menu entry

-

Command line syntax

`--make-target=`*name*

Description

With this option you can overrule the default target name in the make dependencies generated by the options **--preprocess=+make** (**-Em**) and **--dep-file**. The default target name is the basename of the input file, with extension `.o`.

Example

```
cctc --preprocess=+make --make-target=../mytarget.o test.c
```

The compiler generates dependency lines with the default target name `../mytarget.o` instead of `test.o`.

Related information

Control program option **--preprocess=+make** (Generate dependencies for make)

Control program option **--dep-file** (Generate dependencies in a file)

Control program option: --mil-link / --mil-split

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. Enable the option **Build for application wide optimizations (MIL linking)**.

3. Select **Optimize less/Build faster** or **Optimize more/Build slower**.

Command line syntax

```
--mil-link
--mil-split[=file,...]
```

Description

With option **--mil-link** the C compiler links the optimized intermediate representation (MIL) of all input files and MIL libraries specified on the command line in the compiler. The result is one single module that is optimized another time.

Option **--mil-split** does the same as option **--mil-link**, but in addition, the resulting MIL representation is written to a file with the suffix `.mil` and the C compiler also splits the MIL representation and writes it to separate files with suffix `.ms`. One file is written for each input file or MIL library specified on the command line. The `.ms` files are only updated on a change.

With option **--mil-split** you can perform application-wide optimizations during the frontend phase by specifying all modules at once, and still invoke the backend phase one module at a time to reduce the total compilation time. Application wide code compaction is not possible in this case.

Optionally, you can specify another filename for the `.ms` file the C compiler generates. Without an argument, the basename of the C source file is used to create the `.ms` filename. Note that if you specify a filename, you have to specify one filename for every input file.

Note that with both options some extra strict type checking is done that can cause building to fail in a way that is unforeseen and difficult to understand. For example, when you use one of these options in combination with option **--uchar** you might get the following error:

```
ctc E289: ["..\..\..\strlen.c" 14/1] "strlen" redeclared with a different type
ctc I802: ["installation-dir\ctc\include\string.h" 44/17]
        previous declaration of "strlen"
1 errors, 0 warnings
```

This is caused by the fact that the MIL library is built without **--uchar**. You can workaround this problem by rebuilding the MIL libraries.

Build for application wide optimizations (MIL linking) and Optimize less/Build faster

This option is standard MIL linking and splitting. Note that you can control the optimizations to be performed with the optimization settings.

Optimize more/Build slower

When you enable this option, the compiler's frontend does not split the MIL stream in separate modules, but feeds it directly to the compiler's backend, allowing the code compaction to be performed application wide.

Related information

Section 4.1, *Compilation Process*

C compiler option **--mil** / **--mil-split**

Control program option: --mmu-present / --mmu-on

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor Selection** list, select a processor that has an MMU or select a user defined processor.

3. (Optional) Select **C/C++ Compiler » Miscellaneous**.

4. (Optional) Add the option **--mmu-present** to the **Additional options** field.

5. Select **C/C++ Compiler » Code Generation**.

6. Enable the option **Use the MMU if present**.

> For the TriCore processor TC1130, the option **--mmu-present** is set automatically, so you only need to enable the option **Use the MMU if present**. For user defined processors you need to enable them both.

Command line syntax

```
--mmu-present
--mmu-on
```

Description

If the processor you are using has an MMU which is activated, you can tell the compiler to use the MMU. The compiler then will align data naturally. To instruct the compiler to use the MMU, you must set both the option **--mmu-present** and the option **--mmu-on**.

With the option **--mmu-present** you tell the compiler that an MMU is physically present. With the option **--mmu-on** you tell the compiler the MMU is activated. In this case the compiler needs to align data naturally. Both options are necessary.

If you select the TC1130 processor (command line option **--cpu** (**-C**)), the option **--mmu-present** is set automatically. If you are using a target processor with MMU that is not predefined, you need to set this option manually.

Note that the option **--mmu-on** is only required for the TC1130 when the MMU is used on that device. For the C compiler this option provides a workaround for silicon bug CPU_TC.052 Alignment Restrictions for Accesses using PTE-Based Translation, which states that natural alignment must be used for accesses undergoing PTE-based translation which may result in a non-cacheable memory access.

The control program passes the option **--mmu-present** to the assembler also. With this option you can use memory management instructions in the assembly code. When you select this option, the define __MMU__ is set to 1.

Example

To allow the use of memory management unit (MMU) instructions in the assembly code, enter:

```
cctc --mmu-present test.c
```

Related information

Control program option **--cpu** (Select processor)

Control program option **--fp-model=+soft** (Disable hardware floating-point instructions)

Control program option: --new-task

Menu entry

1. In the C/C++ Projects view, right-click on the name of a TriCore project and select **Properties**.

    The Properties dialog appears.

2. In the left pane, select **Project References**.

3. In the right pane, select the MCS project or 8051 project that must be part of the TriCore project and click **OK**.

Command line syntax

--new-task=`core,{obj|out|lib}[,obj|out|lib]...`

You can specify the following cores:

mcs0*n*	MCS core (*n*=0 .. 6)
xc800	8051 core

Description

With this option you can specify to link a task for a specific *core*. Eclipse will call this option when you reference an MCS project or 8051 project from a TriCore project. The first argument is a core name. After the core name a comma separated task input list is specified. The input list is a list of objects, incrementally linked files (`.out`) or library files (`.lib`).

Example

To link objects `foo1.o`, `foo2.o` of MCS core `mcs00` and `foo3.o` and `foo4.o` of MCS core `mcs01` to the TriCore project, enter:

```
cctc --new-task=mcs00,foo1.o,foo2.o --new-task=mcs01,foo3.o,foo4.o mytcproj.c
```

This results in the following options being passed to the linker:

```
--new-task --core=mpe:mcs00 --map-file=mcs00.map foo1.o foo2.o
--new-task --core=mpe:mcs01 --map-file=mcs01.map foo3.o foo4.o
```

Related information

Linker option **--new-task** (Create additional task)

Control program option: --no-default-libraries

Menu entry

1. Select **Linker » Libraries**.

2. Disable the option **Link default libraries**.

Command line syntax

```
--no-default-libraries
```

Description

By default the control program specifies the standard C libraries (C99) and run-time library to the linker. With this option you tell the control program not to specify the standard C libraries and run-time library to the linker.

In this case you must specify the libraries you want to link to the linker with the option **--library=***library_name* or pass the libraries as files on the command line. The control program recognizes the option **--library** (**-l**) as an option for the linker and passes it as such.

Example

```
cctc --no-default-libraries test.c
```

The control program does not specify any libraries to the linker. In normal cases this would result in unresolved externals.

To specify your own libraries (`libc.a`) and avoid unresolved externals:

```
cctc --no-default-libraries --library=c test.c
```

Related information

Control program option **--library** (Link system library)

Section 7.3.1, *How the Linker Searches Libraries*

Control program option: --no-map-file

Menu entry

1. Select **Linker » Map File**.

2. Disable the option **Generate map file**.

Command line syntax

```
--no-map-file
```

Description

By default the control program tells the linker to generate a linker map file.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (`.o`) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

With this option you prevent the generation of a map file.

Related information

-

Control program option: --no-warnings (-w)

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

    The Suppress C compiler warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.

3. Enter the numbers, separated by commas or as a range, of the warnings you want to suppress (for example 537,538). Or you can use the **Add** button multiple times.

4. To suppress all warnings, enable the option **Suppress all warnings**.

Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

Command line syntax

--no-warnings[**=***number*[*-number*],...]

-w[*number*[*-number*],...]

Description

With this option you can suppresses all warning messages for the various tools or specific control program warning messages.

On the command line this option works as follows:

• If you do not specify this option, all warnings are reported.

• If you specify this option but without numbers, all warnings of all tools are suppressed.

• If you specify this option with a number or a range, only the specified control program warnings are suppressed. You can specify the option **--no-warnings=***number* multiple times.

Example

To suppress all warnings for all tools, enter:

```
cctc test.c --no-warnings
```

Related information

Control program option **--warnings-as-errors** (Treat warnings as errors)

Control program option: --option-file (-f)

Menu entry

-

Command line syntax

--option-file=*file*,...

-f *file*,...

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the control program.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** multiple times.

Format of an option file

• Multiple arguments on one line in the option file are allowed.

• To include whitespace in an argument, surround the argument with single or double quotes.

• If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

```
"This has a single quote ' embedded"

'This has a double quote " embedded'

'This has a double quote " and a single quote '"' embedded"
```

• When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \
line"

        -> "This is a continuation line"
```

• It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
--debug-info
--define=DEMO=1
test.c
```

Specify the option file to the control program:

```
cctc --option-file=myoptions
```

This is equivalent to the following command line:

```
cctc —-debug-info --define=DEMO=1 test.c
```

Related information

-

Control program option: --output (-o)

Menu entry

Eclipse always uses the project name as the basename for the output file.

Command line syntax

--output=_file_

-o _file_

Description

By default, the control program generates a file with the same basename as the first specified input file. With this option you specify another name for the resulting absolute object file.

The default output format is ELF/DWARF, but you can specify another output format with option **--format**.

Example

```
cctc test.c prog.c
```

The control program generates an ELF/DWARF object file (default) with the name `test.elf`.

To generate the file `result.elf`:

```
cctc --output=result.elf test.c prog.c
```

Related information

Control program option **--format** (Set linker output format)

Linker option **--output** (Output file)

Linker option **--chip-output** (Generate an output file for each chip)

Control program option: --pass (-W)

Menu entry

1. Select **C/C++ Compiler » Miscellaneous** or **Assembler » Miscellaneous** or **Linker » Miscellaneous**.

2. Add an option to the **Additional options** field.

    *Be aware that the options in the option file are added to the options you have set in the other pages. Only in extraordinary cases you may want to use them in combination. The assembler options are preceded by **-Wa** and the linker options are preceded by **-Wl**. For the C/C++ options you have to do this manually.*

Command line syntax

--pass-assembler=*option*	**-Wa***option*	Pass option directly to the assembler
--pass-c=*option*	**-Wc***option*	Pass option directly to the C compiler
--pass-c++=*option*	**-Wcp***option*	Pass option directly to the C++ compiler
--pass-linker=*option*	**-Wl***option*	Pass option directly to the linker

Description

With this option you tell the control program to call a tool with the specified option. The control program does not use or interpret the option itself, but specifies it directly to the tool which it calls.

Example

To pass the option **--verbose** directly to the linker, enter:

```
cctc --pass-linker=--verbose test.c
```

Related information

-

Control program option: --preprocess (-E) / --no-preprocessing-only

Menu entry

1.   Select **C/C++ Compiler » Preprocessing**.

2.   Enable the option **Store preprocessor output in <file>.pre**.

3.   (Optional) Enable the option **Keep comments in preprocessor output**.

4.   (Optional) Enable the option **Keep #line info in preprocessor output**.

Command line syntax

--preprocess[**=***flags*]

-E[*flags*]

--no-preprocessing-only

You can set the following flags:

+/-comments	**c/C**	keep comments
+/-includes	**i/I**	generate a list of included source files
+/-list	**l/L**	generate a list of macro definitions
+/-make	**m/M**	generate dependencies for make
+/-noline	**p/P**	strip #line source position information

Default: **-ECILMP**

Description

With this option you tell the compiler to preprocess the C source. The C compiler sends the preprocessed output to the file *name*.pre (where *name* is the name of the C source file to compile). Eclipse also compiles the C source.

On the command line, the control program stops after preprocessing. If you also want to compile the C source you can specify the option **--no-preprocessing-only**. In this case the control program calls the compiler twice, once with option **--preprocess** and once for a regular compilation.

With **--preprocess=+comments** you tell the preprocessor to keep the comments from the C source file in the preprocessed output.

With **--preprocess=+includes** the compiler will generate a list of all included source files. The preprocessor output is discarded.

With **--preprocess=+list** the compiler will generate a list of all macro definitions. The preprocessor output is discarded.

With **--preprocess=+make** the compiler will generate dependency lines that can be used in a Makefile. The information is written to a file with extension `.d`. The preprocessor output is discarded. The default target name is the basename of the input file, with the extension `.o`. With the option **--make-target** you can specify a target name which overrules the default target name.

With **--preprocess=+noline** you tell the preprocessor to strip the #line source position information (lines starting with `#line`). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is easier to read.

Example

```
cctc --preprocess=+comments,-make,-noline --no-preprocessing-only test.c
```

The compiler preprocesses the file `test.c` and sends the output to the file `test.pre`. Comments are included but no dependencies are generated and the line source position information is not stripped from the output file. Next, the control program calls the compiler, assembler and linker to create the final object file `test.elf`

Related information

Control program option **--dep-file** (Generate dependencies in a file)

Control program option **--make-target** (Specify target name for **-Em** output)

Control program option: --processors

Menu entry

1. From the **Window** menu, select **Preferences**.

   The Preferences dialog appears.

2. Select **TASKING » TriCore**.

3. Click the **Add** button to add additional processor definition files.

Command line syntax

--processors=*file*

Description

With this option you can specify an additional XML file with processor definitions.

The standard list of supported processors is defined in the file processors.xml. This file defines for each processor its full name (for example, TC1796B), its ID, the base CPU name (for example, tc1796b), the core settings (for example, tc1.3), the MMU and FPU settings and the list of silicon bugs for that processor.

The control program reads the specified *file* after the file processors.xml in the product's etc directory. Additional XML files can override processor definitions made in XML files that are read before.

Multiple **--processors** options are allowed.

Eclipse generates a **--processors** option in the makefiles for each specified XML file.

Example

Specify an additional processor definition file (suppose processors-new.xml contains a new processor TCNEW):

cctc --processors=processors-new.xml --cpu=TCNEW test.c

Related information

Control program option **--cpu** (Select processor)

Control program option **--core** (Select the core)

Control program option: --profile (-p)

Menu entry

1. Select **C/C++ Compiler » Debugging**.

2. Enable or disable **Static profiling**.

3. Enable or disable one or more of the following **Generate profiling information** options (dynamic profiling):

    *   **for block counters** (not in combination with Call graph or Function timers)

    *   **to build a call graph**

    *   **for function counters**

    *   **for function timers**

4. Optionally, when you selected function timers, enable option **Exclude time spent in interrupt functions**.

Note that the more detailed information you request, the larger the overhead in terms of execution time, code size and heap space needed. The option **--debug** does not affect profiling, execution time or code size.

Command line syntax

--profile[**=**`flag`,...]

-p[`flags`]

Use the following option for a predefined set of flags:

--profile=g	**-pg**	Profiling with call graph and function timers. Alias for: **-pBcFStI**

You can set the following flags:

+/-block	**b/B**	block counters
+/-callgraph	**c/C**	call graph
+/-function	**f/F**	function counters
+/-interrupt	**i/I**	exclude interrupt time
+/-static	**s/S**	static profile generation
+/-time	**t/T**	function timers

Default: **-pBCFSTI**

Default (without flags): **-pBCfSTI**

Description

Profiling is the process of collecting statistical data about a running application. With these data you can analyze which functions are called, how often they are called and what their execution time is.

Several methods of profiling exist. One method is *code instrumentation* which adds code to your application that takes care of the profiling process when the application is executed. Another method is *static profiling*.

For an extensive description of profiling refer to Chapter 12, *Profiling*.

You can obtain the following profiling data (see flags above):

Block counters (not in combination with Call graph or Function timers)

This will instrument the code to perform basic block counting. As the program runs, it counts the number of executions of each branch in an if statement, each iteration of a for loop, and so on. Note that though you can combine Block counters with Function counters, this has no effect because Function counters is only a subset of Block counters.

Call graph (not in combination with Block counters)

This will instrument the code to reconstruct the run-time call graph. As the program runs it associates the caller with the gathered profiling data.

Function counters

This will instrument the code to perform function call counting. This is a subset of the basic Block counters.

Function timers (not in combination with Block counters/Function counters)

This will instrument the code to measure the time spent in a function. This includes the time spent in all sub functions (callees). Also the time spent in interrupt functions is taken into account, unless you enable option **Exclude time spent in interrupt functions** (**-pti**).

Static profiling

With this option you do not need to run the application to get profiling results. The compiler generates profiling information at compile time, without adding extra code to your application.

> Note that the more detailed information you request, the larger the overhead in terms of execution time, code size and heap space needed. The option **Generate symbolic debug information** (**--debug**) does not affect profiling, execution time or code size.

The control program automatically specifies the corresponding profiling libraries to the linker.

Example

To generate block count information for the module `test.c` during execution, compile as follows:

```
cctc --profile=+block test.c
```

In this case the control program tells the linker to link the library `libpb.a`.

Related information

Chapter 12, *Profiling*

Control program option: --show-c++-warnings

Menu entry

-

Command line syntax

`--show-c++-warnings`

Description

The C++ compiler may generate a compiled C++ file (`.ic`) that causes warnings during compilation or assembling. With this option you tell the control program to show these warnings. By default C++ warnings are suppressed.

Related information

-

Control program option: --signed-bitfields

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat 'int' bit-fields as signed**.

Command line syntax

`--signed-bitfields`

Description

For bit-fields it depends on the implementation whether a plain `int` is treated as `signed int` or `unsigned int`. By default an `int` bit-field is treated as `unsigned int`. This offers the best performance. With this option you tell the compiler to treat `int` bit-fields as `signed int`. In this case, you can still add the keyword `unsigned` to treat a particular `int` bit-field as `unsigned`.

Related information

C compiler option **--signed-bitfields**

Section 1.1, *Data Types*

Control program option: --silicon-bug

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor selection** list, select a processor.

   *The **CPU problem bypasses and checks** box shows the available workarounds/checks available for the selected processor.*

3. (Optional) Select **Show all CPU problem bypasses and checks**.

4. Click **Select All** or select one or more individual options.

Command line syntax

```
--silicon-bug=arg,...
```

For a list of available arguments refer to the description of option **--silicon-bug** of the compiler and assembler. Depending on the available arguments this option is passed to the compiler and/or assembler.

Description

With this option the control program tells the compiler/assembler/linker to use software workarounds for some CPU functional problems. Please refer to Chapter 18, *CPU Problem Bypasses and Checks* for more information about the individual problems and workarounds.

Example

To enable workarounds for problems CPU_TC.013 and CPU_TC.048, enter:

```
cctc --silicon-bug=cpu-tc013,cpu-tc048 test.c
```

Related information

Chapter 18, *CPU Problem Bypasses and Checks*

C compiler option **--silicon-bug**

Assembler option **--silicon-bug**

Control program option: --software-partitioning-info

Menu entry

1.   Select **C/C++ Compiler » Diagnostics**.

2.   Enable the option **Check software partitioning**.

Eclipse also calls the integrity check tool.

Command line syntax

`--software-partitioning-info`

Description

With this option the compiler adds information about symbol accesses to the output file in
`.debug_tasking_verif` sections and the linker adds information about section access rights to the
output file in `.debug_tasking_verif.rights` sections. This information can be used by the integrity
check utility.

Example

`cctc --software-partitioning-info test.c`

Related information

Section 8.10, *Integrity Check Utility*

Section 8.6.2, *HLL Dump Output Format*

Control program option: --static

Menu entry

-

Command line syntax

`--static`

Description

This option is directly passed to the compiler.

With this option, the compiler treats external definitions at file scope (except for `main`) as if they were declared `static`. As a result, unused functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module.

This option only makes sense when you specify all modules of an application on the command line.

Example

```
cctc --static module1.c module2.c module3.c ...
```

Related information

-

Control program option: --tasking-sfr

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

2. Enable the option **Automatic inclusion of '.sfr' file**.

3. Select **Assembler » Preprocessing**.

4. Enable the option **Automatic inclusion of '.def' file**.

Command line syntax

```
--tasking-sfr
```

Description

By default, the C compiler and assembler do not include a special function register (SFR) file before compiling/assembling.

With this option the compiler includes the register file reg*cpu*.sfr and the assembler includes the file reg*cpu*.def as based on the selected target processor. The compiler and assembler select the SFR file belonging to the target you selected on the **Processor** page (control program option **--cpu**). The control program passes the appropriate **-H** option to the tools.

Example

To generate the file test.elf for the TC1796B processor and automatically include SFR files, enter:

```
cctc --cpu=tc1796b --tasking-sfr -v -t test.c
```

The control program will call the tools as follows:

```
ctc  -D__CPU__=tc1796b -D__CPU_TC1796B__ --core=tc1.3 --fp-model=+float
     -Hsfr/regtc1796b.sfr -o test.src test.c
astc -D__CPU__=tc1796b -D__CPU_TC1796B__ --core=tc1.3
     -Hsfr/regtc1796b.def -o test.o test.src
lktc -o test.elf -dtc1796b.lsl -dextmem.lsl -D__CPU__=tc1796b
     -D__PROC_TC1796B__ --map-file test.o -lcs_fpu -lfp_fpu
     -lrt -Linstall-dir/ctc/lib/tc13
```

Related information

Control program option **--cpu** (Select processor)

Section 1.3.2, *Accessing Hardware from C*

Control program option: --uchar (-u)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat 'char' variables as unsigned**.

Command line syntax

```
--uchar
```

```
-u
```

Description

By default char is the same as specifying signed char. With this option char is the same as unsigned char. This option is passed to both the C++ compiler and the C compiler.

Note that this option can cause conflicts when you use it in combination with MIL linking. With MIL linking some extra strict type checking is done that can cause building to fail in a way that is unforeseen and difficult to understand. For example, when you use option **--mil-link** in combination with option **--uchar** you might get the following error:

```
ctc E289: ["..\..\..\strlen.c" 14/1] "strlen" redeclared with a different type
ctc I802: ["installation-dir\ctc\include\string.h" 44/17]
          previous declaration of "strlen"
1 errors, 0 warnings
```

This is caused by the fact that the MIL library is built without **--uchar**. You can workaround this problem by rebuilding the MIL libraries.

Related information

Section 1.1, *Data Types*

Control program option: --undefine (-U)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**

   The Defined symbols box shows the symbols that are currently defined.

2. To remove a defined symbol, select the symbol in the **Defined symbols** box and click on the **Delete** button.

Command line syntax

--undefine=`macro_name`

-U`macro_name`

Description

With this option you can undefine an earlier defined macro as with `#undef`. This option is for example useful to undefine predefined macros.

The following predefined ISO C standard macros cannot be undefined:

`__FILE__`	current source filename
`__LINE__`	current source line number (int type)
`__TIME__`	hh:mm:ss
`__DATE__`	Mmm dd yyyy
`__STDC__`	level of ANSI standard

The control program passes the option **--undefine** (**-U**) to the compiler.

Example

To undefine the predefined macro `__TASKING__`:

```
cctc --undefine=__TASKING__ test.c
```

Related information

Control program option **--define** (Define preprocessor macro)

Section 1.9, *Predefined Preprocessor Macros*

Control program option: --user-mode

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--user-mode** to the **Additional options** field.

Command line syntax

--user-mode=*mode*

You can specify the following *mode* arguments:

user-0	User-0 unprivileged mode
user-1	User-1 privileged mode
kernel	Kernel/Supervisor mode

Default: **--user-mode=kernel**

Description

With this option you specify the mode (I/O privilege mode) the TriCore runs in: User-0, User-1 or Kernel/Supervisor. The availability of some instructions depends on this mode. Most instructions run in all modes. The instructions `enable` and `disable` run in User-1 or kernel mode only. The instructions `bisr`, `mtcr`, `cachea.i` and `tlb` instructions run in kernel mode only.

Example

To restrict the instructions in the assembly code to User-1 mode, enter:

```
cctc --user-mode=user-1 test.c
```

Related information

C compiler option **--user-mode** (Select user mode)

Assembler option **--user-mode** (Select user mode)

Control program option: --vared-info-section

Menu entry

1. Select **Linker » Patching**.

2. Enable the option **Patch object and library files supplied to the linker**.

3. Specify a **Data reference modification file**.

4. From the **Add info section to patched object files** list, select **Full**, **Compact**, **Executed commands** or **None**.

Command line syntax

--vared-info-section=`flag`

You can specify the following arguments:

compact	Leave out comments and most white space.
executed-commands	Copy only the executed commands.
full	Copy the entire data reference modification file.
no-section	Do not create the `.VarEd_Info` section.

Default: **full**

Description

With this option you can determine what will be copied from the ELF patch data reference modification file into the new section named `.VarEd_Info` when calling the **elfpatch** utility, or you can choose not to generate the `.VarEd_Info` section at all. By default the entire data reference modification file is copied into the `.VarEd_Info` section.

Example

To read data reference modifications from file `myproject.drm`, and create a compact version in a `.VarEd_Info` section, enter:

```
cctc test.c -Ctc27x --data-reference-modification-file=myproject.drm
          --vared-info-section=compact mylib.a
```

Related information

Section 8.7, *ELF Patch Utility*

Control program option **--data-reference-modification-file**

Control program option: --verbose (-v)

Menu entry

1. Select **Global Options**.

2. Enable the option **Verbose mode of control program**.

Command line syntax

`--verbose`

`-v`

Description

With this option you put the control program in verbose mode. The control program performs its tasks while it prints the steps it performs to stdout.

Related information

Control program option **--dry-run** (Verbose output and suppress execution)

Control program option: --version (-V)

Menu entry

-

Command line syntax

`--version`

`-V`

Description

Display version information. The control program ignores all other options or input files.

Related information

-

Control program option: --warnings-as-errors

Menu entry

1. Select **Global Options**.

2. Enable the option **Treat warnings as errors**.

Command line syntax

`--warnings-as-errors``[=`*`number`*`[`*`-number`*`],...]`

Description

If one of the tools encounters an error, it stops processing the file(s). With this option you tell the tools to treat warnings as errors or treat specific control program warning messages as errors:

• If you specify this option but without numbers, all warnings are treated as errors.

• If you specify this option with a number or a range, only the specified control program warnings are treated as an error. You can specify the option **--warnings-as-errors=**_number_ multiple times.

Use one of the **--pass-**_tool_ options to pass this option directly to a tool when a specific warning for that tool must be treated as an error. For example, use **--pass-c=--warnings-as-errors=**_number_ to treat a specific C compiler warning as an error.

Related information

Control program option **--no-warnings** (Suppress some or all warnings)

Control program option **--pass** (Pass option to tool)

10.7. Make Utility Options

When you build a project in Eclipse, Eclipse generates a makefile and uses the make utility **mktc** to build all your files. However, you can also use the make utility directly from the command line to build your project.

The invocation syntax is:

mktc [*option*...] [*target*...] [*macro=def*]

This section describes all options for the make utility. The make utility is a command line tool so there are no equivalent options in Eclipse.

For detailed information about the make utility and using makefiles see Section 8.2, *Make Utility mktc*.

Defining Macros

Command line syntax

macro_name[**=***macro_definition*]

Description

With this argument you can define a macro and specify it to the make utility.

A macro definition remains in existence during the execution of the makefile, even when the makefile recursively calls the make utility again. In the recursive call, the macro acts as an environment variable. This means that it is overruled by definitions in the recursive call. Use the option **-e** to prevent this.

You can specify as many macros as you like. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the make utility with the option **-m**) *file*.

Defining macros on the command line is, for example, useful in combination with conditional processing as shown in the example below.

Example

Consider the following makefile with conditional rules to build a demo program and a real program:

```
ifdef DEMO        # the value of DEMO is of no importance
   real.elf : demo.o main.o
             ltc demo.o main.o -lc -lfp -lrt
else
   real.elf : real.o main.o
             ltc real.o main.o -lc -lfp -lrt
endif
```

You can now use a macro definition to set the DEMO flag:

```
mktc real.elf DEMO=1
```

In both cases the absolute object file real.elf is created but depending on the DEMO flag it is linked with demo.o or with real.o.

Related information

Make utility option **-e** (Environment variables override macro definitions)

Make utility option **-m** (Name of invocation file)

Make utility option: -?

Command line syntax

`-?`

Description

Displays an overview of all command line options.

Example

The following invocation displays a list of the available command line options:

```
mktc -?
```

Related information

-

Make utility option: -a

Command line syntax

`-a`

Description

Normally the make utility rebuilds only those files that are out of date. With this option you tell the make utility to rebuild *all* files, without checking whether they are out of date.

Example

```
mktc -a
```

Rebuilds all your files, regardless of whether they are out of date or not.

Related information

-

Make utility option: -c

Command line syntax

`-c`

Description

Eclipse uses this option when you create sub-projects. In this case the make utility calls another instance of the make utility for the sub-project. With the option **-c**, the make utility runs as a child process of the current make.

The option **-c** overrules the option **-err**.

Example

```
mktc -c
```

The make utility runs its commands as a child processes.

Related information

Make utility option **-err** (Redirect error message to file)

Make utility option: -D / -DD

Command line syntax

-D
-DD

Description

With the option **-D** the make utility prints every line of the makefile to standard output as it is read by **mktc**.

With the option **-DD** not only the lines of the makefile are printed but also the lines of the `mktc.mk` file (implicit rules).

Example

```
mktc -D
```

Each line of the makefile that is read by the make utility is printed to standard output (usually your screen).

Related information

-

Make utility option: -d/ -dd

Command line syntax

`-d`
`-dd`

Description

With the option **-d** the make utility shows which files are out of date and thus need to be rebuild. The option **-dd** gives more detail than the option **-d**.

Example

```
mktc -d
```

Shows which files are out of date and rebuilds them.

Related information

-

Make utility option: -e

Command line syntax

-e

Description

If you use macro definitions, they may overrule the settings of the environment variables. With the option **-e**, the settings of the environment variables are used even if macros define otherwise.

Example

```
mktc -e
```

The make utility uses the settings of the environment variables regardless of macro definitions.

Related information

-

Make utility option: -err

Command line syntax

-err *file*

Description

With this option the make utility redirects error messages and verbose messages to a specified file.

With the option **-s** the make utility only displays error messages.

Example

```
mktc -err error.txt
```

The make utility writes messages to the file error.txt.

Related information

Make utility option **-s** (Do not print commands before execution)

Make utility option **-c** (Run as child process)

Make utility option: -f

Command line syntax

-f *my_makefile*

Description

By default the make utility uses the file makefile to build your files.

With this option you tell the make utility to use the specified file instead of the file makefile. Multiple **-f** options act as if all the makefiles were concatenated in a left-to-right order.

If you use '-' instead of a makefile name it means that the information is read from stdin.

Example

```
mktc -f mymake
```

The make utility uses the file mymake to build your files.

Related information

-

Make utility option: -G

Command line syntax

-G *path*

Description

Normally you must call the make utility from the directory where your makefile and other files are stored.

With the option **-G** you can call the make utility from within another directory. The *path* is the path to the directory where your makefile and other files are stored and can be absolute or relative to your current directory.

Example

Suppose your makefile and other files are stored in the directory `..\myfiles`. You can call the make utility, for example, as follows:

```
mktc -G ..\myfiles
```

Related information

-

Make utility option: -i

Command line syntax

-i

Description

When an error occurs during the make process, the make utility exits with a certain exit code.

With the option **-i**, the make utility exits without an error code, even when errors occurred.

Example

```
mktc -i
```

The make utility exits without an error code, even when an error occurs.

Related information

-

Make utility option: -K

Command line syntax

`-K`

Description

With this option the make utility keeps temporary files it creates during the make process. The make utility stores temporary files in the directory that you have specified with the environment variable TMPDIR or in the default 'temp' directory of your system when the TMPDIR environment variable is not specified.

Example

```
mktc -K
```

The make utility preserves all temporary files.

Related information

-

Make utility option: -k

Command line syntax

`-k`

Description

When during the make process the make utility encounters an error, it stops rebuilding your files.

With the option **-k**, the make utility only stops building the target that produced the error. All other targets defined in the makefile are built.

Example

```
mktc -k
```

If the make utility encounters an error, it stops building the current target but proceeds with the other targets that are defined in the makefile.

Related information

Make utility option **-S** (Undo the effect of **-k**)

Make utility option: -m

Command line syntax

-m *file*

Description

Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the make utility.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **-m** multiple times.

If you use '-' instead of a filename it means that the options are read from stdin.

Format of an option file

- Multiple arguments on one line in the option file are allowed.

- To include whitespace in an argument, surround the argument with single or double quotes.

- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

  ```
  "This has a single quote ' embedded"

  'This has a double quote " embedded'

  'This has a double quote " and a single quote '"' embedded"
  ```

  Note that adjacent strings are concatenated.

- When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

  ```
  "This is a continuation \
  line"

          -> "This is a continuation line"
  ```

- It is possible to nest command line files up to 25 levels.

Example

Suppose the file myoptions contains the following lines:

```
-k
-err errors.txt
test.elf
```

Specify the option file to the make utility:

```
mktc -m myoptions
```

This is equivalent to the following command line:

```
mktc -k -err errors.txt test.elf
```

Related information

-

Make utility option: -n

Command line syntax

`-n`

Description

With this option you tell the make utility to perform a dry run. The make utility shows what it would do but does not actually perform these tasks.

This option is for example useful to quickly inspect what would happen if you call the make utility.

Example

```
mktc -n
```

The make utility does not perform any tasks but displays what it would do if called without the option **-n**.

Related information

Make utility option **-s** (Do not print commands before execution)

Make utility option: -p

Command line syntax

`-p`

Description

Normally, if a command in a target rule in a makefile returns an error or when the target construction is interrupted, the make utility removes that target file. With this option you tell the make utility to make all target files precious. This means that all dependency files are never removed.

Example

```
mktc -p
```

The make utility never removes target dependency files.

Related information

Special target `.PRECIOUS` in Section 8.2.2.1, *Targets and Dependencies*

Make utility option: -q

Command line syntax

`-q`

Description

With this option the make utility does not perform any tasks but only returns an exit code. A zero status indicates that all target files are up to date, a non-zero status indicates that some or all target files are out of date.

Example

```
mktc -q
```

The make utility only returns an error code that indicates whether all target files are up to date or not. It does not rebuild any files.

Related information

-

Make utility option: -r

Command line syntax

`-r`

Description

When you call the make utility, it first reads the implicit rules from the file `mktc.mk`, then it reads the makefile with the rules to build your files. (The file `mktc.mk` is located in the `\etc` directory of the toolset.)

With this option you tell the make utility not to read `mktc.mk` and to rely fully on the make rules in the makefile.

Example

`mktc -r`

The make utility does not read the implicit make rules in `mktc.mk`.

Related information

-

Make utility option: -S

Command line syntax

-S

Description

With this option you cancel the effect of the option **-k**. This is only necessary in a recursive make where the option **-k** might be inherited from the top-level make via MAKEFLAGS or if you set the option **-k** in the environment variable MAKEFLAGS.

With this option you tell the make utility not to read `mktc.mk` and to rely fully on the make rules in the makefile.

Example

```
mktc -S
```

The effect of the option **-k** is cancelled so the make utility stops with the make process after it encounters an error.

The option **-k** in this example may have been set with the environment variable MAKEFLAGS or in a recursive call to **mktc** in the makefile.

Related information

Make utility option **-k** (On error, abandon the work for the current target only)

Make utility option: -s

Command line syntax

`-s`

Description

With this option you tell the make utility to perform its tasks without printing the commands it executes. Error messages are normally printed.

Example

```
mktc -s
```

The make utility rebuilds your files but does not print the commands it executes during the make process.

Related information

Make utility option **-n** (Perform a dry run)

Make utility option: -t

Command line syntax

`-t`

Description

With this option you tell the make utility to *touch* the target files, bringing them up to date, rather than performing the rules to rebuild them.

Example

```
mktc -t
```

The make utility updates out-of-date files by giving them a new date and time stamp. The files are not actually rebuild.

Related information

-

Make utility option: -time

Command line syntax

-time

Description

With this option you tell the make utility to display the current date and time on standard output.

Example

```
mktc -time
```

The make utility displays the current date and time and updates out-of-date files.

Related information

-

Make utility option: -V

Command line syntax

-V

Description

Display version information. The make utility ignores all other options or input files.

Example

```
mktc -V
```

The make utility displays the version information but does not perform any tasks.

```
TASKING VX-toolset for TriCore: program builder    vx.yrz Build nnn
Copyright 2004-year Altium BV                      Serial# 00000000
```

Related information

-

Make utility option: -W

Command line syntax

-W *target*

Description

With this option the make utility considers the specified target file always as up to date and will not rebuild it.

Example

```
mktc -W test.elf
```

The make utility rebuilds out of date targets in the makefile except the file test.elf which is considered now as up to date.

Related information

-

Make utility option: -w

Command line syntax

`-w`

Description

With this option the make utility sends error messages and verbose messages to standard output. Without this option, the make utility sends these messages to standard error.

This option is only useful on UNIX systems.

Example

```
mktc -w
```

The make utility sends messages to standard out instead of standard error.

Related information

-

Make utility option: -x

Command line syntax

`-x`

Description

With this option the make utility shows extended error messages. Extended error messages give more detailed information about the exit status of the make utility after errors.

Example

```
mktc -x
```

If errors occur, the make utility gives extended information.

Related information

-

10.8. Parallel Make Utility Options

When you build a project in Eclipse, Eclipse generates a makefile and uses the make utility **amk** to build all your files. However, you can also use the make utility directly from the command line to build your project.

The invocation syntax is:

amk [*option*...] [*target*...] [*macro*=*def*]

This section describes all options for the parallel make utility.

For detailed information about the parallel make utility and using makefiles see Section 8.3, *Make Utility amk*.

Parallel make utility option: --always-rebuild (-a)

Command line syntax

`--always-rebuild`

`-a`

Description

Normally the make utility rebuilds only those files that are out of date. With this option you tell the make utility to rebuild *all* files, without checking whether they are out of date.

Example

```
amk -a
```

Rebuilds all your files, regardless of whether they are out of date or not.

Related information

-

Parallel make utility option: --change-dir (-G)

Command line syntax

--change-dir=`path`

-G `path`

Description

Normally you must call the make utility from the directory where your makefile and other files are stored.

With the option **-G** you can call the make utility from within another directory. The *path* is the path to the directory where your makefile and other files are stored and can be absolute or relative to your current directory.

The macro `SUBDIR` is defined with the value of *path*.

Example

Suppose your makefile and other files are stored in the directory `..\myfiles`. You can call the make utility, for example, as follows:

```
amk -G ..\myfiles
```

Related information

-

Parallel make utility option: --diag

Command line syntax

--diag=[*format*:]{**all** | *nr*,...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

Example

To display an explanation of message number 169, enter:

```
amk --diag=169
```

This results in the following message and explanation:

```
F169: target '%s' returned exit code %d

An error occured while executing one of the commands
of the target, and -k option is not specified.
```

To write an explanation of all errors and warnings in HTML format to file `amkerrors.html`, use redirection and enter:

```
amk --diag=html:all > amkerrors.html
```

Related information

-

Parallel make utility option: --dry-run (-n)

Command line syntax

`--dry-run`

`-n`

Description

With this option you tell the make utility to perform a dry run. The make utility shows what it would do but does not actually perform these tasks.

This option is for example useful to quickly inspect what would happen if you call the make utility.

Example

```
amk -n
```

The make utility does not perform any tasks but displays what it would do if called without the option **-n**.

Related information

Parallel make utility option **-s** (Do not print commands before execution)

Parallel make utility option: --help (-? / -h)

Command line syntax

--help[**=***item*]

-h

-?

You can specify the following arguments:

      options   Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

The following invocations all display a list of the available command line options:

```
amk -?
amk --help
```

To see a detailed description of the available options, enter:

```
amk --help=options
```

Related information

-

Parallel make utility option: --jobs (-j) / --jobs-limit (-J)

Menu

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, select **C/C++ Build**.

    In the right pane the C/C++ Build page appears.

3. On the Behaviour tab, select **Use parallel build**.

4. You can specify the number of parallel jobs, or you can use an optimal number of jobs. In the last case, **amk** will fork as many jobs in parallel as cores are available.

Command line syntax

```
--jobs[=number]
-j[number]

--jobs-limit[=number]
-J[number]
```

Description

When these options you can limit the number of parallel jobs. The default is 1. Zero means no limit. When you omit the *number*, **amk** uses the number of cores detected.

Option **-J** is the same as **-j**, except that the number of parallel jobs is limited by the number of cores detected.

Example

```
amk -j3
```

Limit the number of parallel jobs to 3.

Related information

-

Parallel make utility option: --keep-going (-k)

Command line syntax

`--keep-going`

`-k`

Description

When during the make process the make utility encounters an error, it stops rebuilding your files.

With the option **-k**, the make utility only stops building the target that produced the error. All other targets defined in the makefile are built.

Example

```
amk -k
```

If the make utility encounters an error, it stops building the current target but proceeds with the other targets that are defined in the makefile.

Related information

-

Parallel make utility option: --list-targets (-l)

Command line syntax

`--list-targets`

`-l`

Description

With this option, the make utility lists all "primary" targets that are out of date.

Example

```
amk -l
list of targets
```

Related information

-

Parallel make utility option: --makefile (-f)

Command line syntax

--makefile=*my_makefile*

-f *my_makefile*

Description

By default the make utility uses the file makefile to build your files.

With this option you tell the make utility to use the specified file instead of the file makefile. Multiple **-f** options act as if all the makefiles were concatenated in a left-to-right order.

If you use '-' instead of a makefile name it means that the information is read from stdin.

Example

```
amk -f mymake
```

The make utility uses the file mymake to build your files.

Related information

-

Parallel make utility option: --no-warnings (-w)

Command line syntax

--no-warnings[**=**_number_,...]

-w[_number_,...]

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.

- If you specify this option but without numbers, all warnings are suppressed.

- If you specify this option with a number, only the specified warning is suppressed. You can specify the option **--no-warnings=**_number_ multiple times.

Example

To suppress warnings 751 and 756, enter:

```
amk --no-warnings=751,756
```

Related information

Parallel make utility option **--warnings-as-errors** (Treat warnings as errors)

Parallel make utility option: --silent (-s)

Command line syntax

`--silent`

`-s`

Description

With this option you tell the make utility to perform its tasks without printing the commands it executes. Error messages are normally printed.

Example

```
amk -s
```

The make utility rebuilds your files but does not print the commands it executes during the make process.

Related information

Parallel make utility option **-n** (Perform a dry run)

Parallel make utility option: --version (-V)

Command line syntax

`--version`

`-V`

Description

Display version information. The make utility ignores all other options or input files.

Related information

-

Parallel make utility option: --warnings-as-errors

Command line syntax

`--warnings-as-errors`[`=`*number*`,...`]

Description

If the make utility encounters an error, it stops. When you use this option without arguments, you tell the make utility to treat all warnings as errors. This means that the exit status of the make utility will be non-zero after one or more warnings. As a consequence, the make utility now also stops after encountering a warning.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

Parallel make utility option **--no-warnings** (Suppress some or all warnings)

10.9. Archiver Options

The archiver and library maintainer **artc** is a tool to build library files and it offers the possibility to replace, extract and remove modules from an existing library.

The invocation syntax is:

artc *key_option* [*sub_option*...] *library* [*object_file*]

This section describes all options for the archiver. Some suboptions can only be used in combination with certain key options. They are described together. Suboptions that can always be used are described separately.

For detailed information about the archiver, see Section 8.5, *Archiver*.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Overview of the options of the archiver utility

The following archiver options are available:

Description	Option	Sub-option
Main functions (key options)		
Replace or add an object module	**-r**	**-a -b -c -n -u -v**
Extract an object module from the library	**-x**	**-o -v**
Delete object module from library	**-d**	**-v**
Move object module to another position	**-m**	**-a -b -v**
Print a table of contents of the library	**-t**	**-s0 -s1**
Print object module to standard output	**-p**	
Sub-options		
Append or move new modules after existing module *name*	**-a** *name*	
Append or move new modules before existing module *name*	**-b** *name*	
Suppress the message that is displayed when a new library is created.	**-c**	
Create a new library from scratch	**-n**	
Preserve last-modified date from the library	**-o**	
Print symbols in library modules	**-s{0|1}**	
Replace only newer modules	**-u**	
Verbose	**-v**	

Description	Option	Sub-option
Miscellaneous		
Display options	**-?**	
Display description of one or more diagnostic messages	**--diag**	
Display version header	**-V**	
Read options from *file*	**-f** *file*	
Suppress warnings above level *n*	**-w***n*	

Archiver option: --diag

Command line syntax

--diag=[*format*:]{**all** | *msg*[*-msg*],...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. The archiver does not perform any actions. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas, or you can specify a range.

Example

To display an explanation of message number 102, enter:

```
artc --diag=102
```

This results in the following message and explanation:

```
F102: cannot create "<file>"
```

```
The output file or a temporary file could not be created. Check if you have
sufficient disk space and if you have write permissions for the specified
file.
```

To write an explanation of all errors and warnings in HTML format to file `arerrors.html`, use redirection and enter:

```
artc --diag=html:all > arerrors.html
```

Related information

-

Archiver option: --delete (-d)

Command line syntax

--delete [**--verbose**]

-d [**-v**]

Description

Delete the specified object modules from a library. With the suboption **--verbose** (**-v**) the archiver shows which files are removed.

    --verbose   **-v**   Verbose: the archiver shows which files are removed.

Example

```
artc --delete mylib.a obj1.o obj2.o
```

The archiver deletes `obj1.o` and `obj2.o` from the library `mylib.a`.

```
artc -d -v mylib.a obj1.o obj2.o
```

The archiver deletes `obj1.o` and `obj2.o` from the library `mylib.a` and displays which files are removed.

Related information

-

Archiver option: --dump (-p)

Command line syntax

`--dump`

`-p`

Description

Print the specified object module(s) in the library to standard output.

This option is only useful when you redirect or pipe the output to other files or tools that serve your own purposes. Normally you do not need this option.

Example

```
artc --dump mylib.a obj1.o > file.o
```

The archiver prints the file `obj1.o` to standard output where it is redirected to the file `file.o`. The effect of this example is very similar to extracting a file from the library but in this case the 'extracted' file gets another name.

Related information

-

Archiver option: --extract (-x)

Command line syntax

--extract [**--modtime**] [**--verbose**]

-x [**-o**] [**-v**]

Description

Extract an existing module from the library.

--modtime	**-o**	Give the extracted object module the same date as the last-modified date that was recorded in the library. Without this suboption it receives the last-modified date of the moment it is extracted.
--verbose	**-v**	Verbose: the archiver shows which files are extracted.

Example

To extract the file obj1.o from the library mylib.a:

```
artc --extract mylib.a obj1.o
```

If you do not specify an object module, all object modules are extracted:

```
artc -x mylib.a
```

Related information

-

Archiver option: --help (-?)

Command line syntax

--help[**=***item*]

-?

You can specify the following argument:

      options   Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

The following invocations all display a list of the available command line options:

```
artc -?
artc --help
artc
```

To see a detailed description of the available options, enter:

```
artc --help=options
```

Related information

-

Archiver option: --move (-m)

Command line syntax

--move [**-a** *posname*] [**-b** *posname*]

-m [**-a** *posname*] [**-b** *posname*]

Description

Move the specified object modules to another position in the library.

The ordering of members in a library can make a difference in how programs are linked if a symbol is defined in more than one member.

By default, the specified members are moved to the end of the archive. Use the suboptions **-a** or **-b** to move them to a specified place instead.

--after=*posname*	**-a** *posname*	Move the specified object module(s) after the existing module *posname*.
--before=*posname*	**-b** *posname*	Move the specified object module(s) before the existing module *posname*.

Example

Suppose the library `mylib.a` contains the following objects (see option **--print**):

```
obj1.o
obj2.o
obj3.o
```

To move `obj1.o` to the end of `mylib.a`:

```
artc --move mylib.a obj1.o
```

To move `obj3.o` just before `obj2.o`:

```
artc -m -b obj3.o mylib.a obj2.o
```

The library `mylib.a` after these two invocations now looks like:

```
obj3.o
obj2.o
obj1.o
```

Related information

Archiver option **--print** (**-t**) (Print library contents)

Archiver option: --option-file (-f)

Command line syntax

--option-file=*file*

-f *file*

Description

Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the archiver.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** (**-f**) multiple times.

If you use '-' instead of a filename it means that the options are read from stdin.

Format of an option file

- Multiple arguments on one line in the option file are allowed.

- To include whitespace in an argument, surround the argument with single or double quotes.

- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

  ```
  "This has a single quote ' embedded"

  'This has a double quote " embedded'

  'This has a double quote " and a single quote '"' embedded"
  ```

- When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

  ```
  "This is a continuation \
  line"

          -> "This is a continuation line"
  ```

- It is possible to nest command line files up to 25 levels.

Example

Suppose the file myoptions contains the following lines:

```
-x mylib.a obj1.o
-w5
```

Specify the option file to the archiver:

```
artc --option-file=myoptions
```

This is equivalent to the following command line:

```
artc -x mylib.a obj1.o -w5
```

Related information

-

Archiver option: --print (-t)

Command line syntax

--print [**--symbols=0**|**1**]

-t [**-s0**|**-s1**]

Description

Print a table of contents of the library to standard output. With the suboption **-s0** the archiver displays all symbols per object file.

--symbols=0	**-s0**	Displays per object the name of the object itself and all symbols in the object.
--symbols=1	**-s1**	Displays the symbols of all object files in the library in the form *library_name*:*object_name*:*symbol_name*

Example

```
artc --print mylib.a
```

The archiver prints a list of all object modules in the library mylib.a:

```
artc -t -s0 mylib.a
```

The archiver prints per object all symbols in the library. For example:

```
cstart.o
   symbols:
        _START
        __init_sp
        _start
        _endinit_clear
        _endinit_set
cinit.o
   symbols:
        _c_init
```

Related information

-

Archiver option: --replace (-r)

Command line syntax

--replace [**--after=**_posname_] [**--before=**_posname_]
       [**--create**] [**--new**] [**--newer-only**] [**--verbose**]

-r [**-a** _posname_] [**-b** _posname_][**-c**] [**-n**] [**-u**] [**-v**]

Description

You can use the option **--replace** (**-r**) for several purposes:

• Adding new objects to the library

• Replacing objects in the library with the same object of a newer date

• Creating a new library

The option **--replace** (**-r**) normally _adds_ a new module to the library. However, if the library already contains a module with the specified name, the existing module is _replaced_. If you specify a library that does not exist, the archiver creates a new library with the specified name.

If you add a module to the library without specifying the suboption **-a** or **-b**, the specified module is added at the end of the archive. Use the suboptions **-a** or **-b** to insert them after/before a specified place instead.

--after=_posname_	**-a** _posname_	Insert the specified object module(s) after the existing module _posname_.
--before=_posname_	**-b** _posname_	Insert the specified object module(s) before the existing module _posname_.
--create	**-c**	Suppress the message that is displayed when a new library is created.
--new	**-n**	Create a new library from scratch. If the library already exists, it is overwritten.
--newer-only	**-u**	Insert the specified object module only if it is newer than the module in the library.
--verbose	**-v**	Verbose: the archiver shows which files are replaced.

The suboptions **-a** or **-b** have no effect when an object is added to the library.

Example

Suppose the library `mylib.a` contains the following object (see option **--print**):

`obj1.o`

To add `obj2.o` to the end of `mylib.a`:

`artc --replace mylib.a obj2.o`

To insert `obj3.o` just before `obj2.o`:

```
artc -r -b obj2.o mylib.a obj3.o
```

The library `mylib.a` after these two invocations now looks like:

```
obj1.o
obj3.o
obj2.o
```

Creating a new library

To *create a new library file*, add an object file and specify a library that does not yet exist:

```
artc --replace newlib.a obj1.o
```

The archiver creates the library `newlib.a` and adds the object `obj1.o` to it.

To *create a new library file and overwrite an existing library*, add an object file and specify an existing library with the supoption **--new** (**-n**):

```
artc -r -n mylib.a obj1.o
```

The archiver overwrites the library `mylib.a` and adds the object `obj1.o` to it. The new library `mylib.a` only contains `obj1.o`.

Related information

Archiver option **--print** (**-t**) (Print library contents)

Archiver option: --version (-V)

Command line syntax

`--version`

`-V`

Description

Display version information. The archiver ignores all other options or input files.

Example

```
artc -V
```

The archiver displays the version information but does not perform any tasks.

```
TASKING VX-toolset for TriCore: ELF archiver    vx.yrz Build nnn
Copyright 2002-year Altium BV                   Serial# 00000000
```

Related information

-

Archiver option: --warning (-w)

Command line syntax

--warning=_level_

-w_level_

Description

With this suboption you tell the archiver to suppress all warnings above the specified level. The level is a number between 0 - 9.

The level of a message is printed between parentheses after the warning number. If you do not use the **-w** option, the default warning level is 8.

Example

To suppress warnings above level 5:

```
artc --extract --warning=5 mylib.a obj1.o
```

Related information

-

10.10. HLL Object Dumper Options

The high level language (HLL) dumper **hldumptc** is a program to dump information about an absolute object file (`.elf`).

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+***longflag*. To switch a flag off, use an uppercase letter or a **-***longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
hldumptc -FdhMsy test.elf
hldumptc --dump-format=+dump,+hllsymbols,-modules,+sections,+symbols test.elf
```

When you do not specify an option, a default value may become active.

HLL object dumper option: --call-graph-elf-mode

Command line syntax

```
--call-graph-elf-mode
```

Description

With this option you can force the call graph to use the ELF symbols instead of the DWARF debug info, for example when dumping from an assembly function.

Related information

Section 8.6.2, *HLL Dump Output Format*

HLL object dumper option: --call-graph-root

Command line syntax

```
--call-graph-root=function
```

Description

With this option you can specify the address or function name where to start the call graph. By default, the call graph starts with main().

When you dump the call graph for an AURIX application, by default only the shared functions are analyzed. When the call graph for the local RAM (CPU0,1,2...) are to be dumped then it is required to specify the option **--copy-table** too. The dumper uses the copy table information to dump local code.

Example

To start the call graph from printf() instead of main(), enter:

```
hldumptc --call-graph-root=printf -F3 hello.elf
```

The call graph looks something like this:

```
+-- 0x8000077a printf
    |
    +-- 0x8000034e _doprint
        |
        +-- 0x8000051c _io_putc
        |   |
        |   +-- 0x80000746 fputc
        |       |
        |       +-- 0x80000428 _flsbuf
        |           |
        |           +-- 0x80000502 _host_write
        |           |   |
        |           |   +-- 0x800002b2 _dbg_trap_tc
        |           |       |
        |           |       +-- 0x80000220 _dbg_cacheawi
        |           |       |
        |           |       +-- 0x80000220 _dbg_cacheawi
        |           |       |
        |           |       +-- 0x80000220 _dbg_cacheawi
        |           |       |
        |           |       +-- 0x80000220 _dbg_cacheawi
        |           |       |
        |           |       +-- 0x8000000e _dbg_trap
        |           |
        |           +-- 0x800003be _fflush
        |           |   |
        |           |   +-- 0x80000502 _host_write *
        |           |   |
```

```
|               |    +-- 0x800004e6 _host_lseek
|               |        |
|               |        +-- 0x800002b2 _dbg_trap_tc *
|               |
|          +-- 0x80000502 _host_write *
|
+-- 0x8000051c _io_putc *
```

Related information

Section 8.6.2, *HLL Dump Output Format*

HLL object dumper option: --class (-c)

Command line syntax

--class[**=***class*]

-c[*class*]

You can specify one of the following classes:

all	**a**	Dump contents of all sections.
code	**c**	Dump contents of code sections.
data	**d**	Dump contents of data sections.

Default: **--class=all**

Description

With this option you can restrict the output to code or data only. This option affects all parts of the output, except the module list. The effect is listed in the following table.

Output part	Effect of --class
Module list	Not restricted
Section list	Only lists sections of the specified class
Section dump	Only dumps the contents of the sections of the specified class
HLL symbol table	Only lists symbols of the specified class
Assembly level symbol table	Only lists symbols defined in sections of the specified class
Note sections	Not restricted

By default all sections are included.

Related information

Section 8.6.2, *HLL Dump Output Format*

HLL object dumper option: --copy-table

Command line syntax

`--copy-table`

Description

With this option the HLL object dumper attempts to translate the specified code address to the destination address of a copy table copy command during disassembly. For example for the TriCore PCP.

When you dump the call graph for an AURIX application and the call graph for the local RAM (CPU0,1,2...) are to be dumped then it is required to specify the option **--copy-table** too. The dumper uses the copy table information to dump local code.

Related information

-

HLL object dumper option: --core-list

Command line syntax

`--core-list`

Description

With this option you can list the TriCore cores used in the application. The compiler scans the copy table for destination addresses of the core local RAMs. In case the address ranges for segment 0x7, 0x6, etc. are found then this means that the core is used (0x7 = CPU0, 0x6 = CPU1, etc.) .

This option is only valid for AURIX derivatives. If a non-AURIX ELF file is passed or the segments 0x0..0x7 are not found in the copy table (as destination) then "Core(s): -" is reported.

Example

To dump the list of used TriCore cores, enter:

```
hldumptc --core-list hello.elf

----------- TriCore core list (based on copy table) ----------

Core(s): CPU0, CPU1
```

Related information

-

HLL object dumper option: --diag

Command line syntax

--diag=[*format*:]{**all** | *msg*[*-msg*],...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. The HLL object dumper does not process any files. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas, or you can specify a range.

Example

To display an explanation of message number 101, enter:

```
hldumptc --diag=101
```

This results in the following message and explanation:

```
F101: cannot create "<file>"

The output file or a temporary file could not be created.
Check if you have sufficient disk space and if you have
write permissions for the specified file.
```

To write an explanation of all errors and warnings in HTML format to file `hldumperrors.html`, use redirection and enter:

```
hldumptc --diag=html:all > hldumperrors.html
```

Related information

-

HLL object dumper option: --disassembly-intermix (-i)

Command line syntax

--disassembly-intermix[**=**_flag_]

-i[_flag_]

You can specify the following format flags:

+/-single-line **s/S** Force the insert to be limited to the first preceding source line.

Default: **--disassembly-intermix=S**

Description

With this option the disassembly is intermixed with HLL source code. The source is searched for as described with option **--source-lookup-path**

The **+single-line** sub-option forces the insert to be limited to the first preceding source line. With the **-single-line** sub-option all source lines that belong to the address are prefixed. For example comments are thus also visible. This is the default.

Example

hldumptc --disassembly-intermix --source-lookup-path=c:\mylib\src hello.elf

Related information

HLL object dumper option **--source-lookup-path**

HLL object dumper option: --disassembly-without-encoding (-r)

Command line syntax

`--disassembly-without-encoding`

`-r`

Description

With this option the address and encoding are not part of the disassembly of a code section. This is useful when you only want the disassembly part.

Example

```
hldumptc -F2 hello.elf

----------- Section dump ----------

                              .sdecl '.text.hello.main', CODE AT 0x80000766
                              .sect  '.text.hello.main'
80000766 20 08       main:    sub.a       sp,#0x8
80000768 85 df 00 08          ld.a        a15,world
8000076c f4 af                st.a        [sp],a15
8000076e 91 00 00 48          movh.a      a4,#0x8000
80000772 d9 44 60 e0          lea         a4,[a4]0x7a0
80000776 1d 00 02 00          j           printf

hldumptc -F2 --disassembly-without-encoding hello.elf

----------- Section dump ----------

       .sdecl '.text.hello.main', CODE AT 0x80000766
       .sect  '.text.hello.main'
main:  sub.a       sp,#0x8
       ld.a        a15,world
       st.a        [sp],a15
       movh.a      a4,#0x8000
       lea         a4,[a4]0x7a0
       j           printf
```

Related information

-

HLL object dumper option: --dump-format (-F)

Command line syntax

--dump-format[**=***flag*,...]

-F[*flag*]...

You can specify the following format flags:

+/-accesses	**a/A**	Dump data accesses per function.
+/-callgraph	**c/C**	Dump the call graph of the application.
+/-dump	**d/D**	Dump the contents of the sections in the object file. Code sections can be disassembled, data sections are dumped.
+/-debug-control-flow	**f/F**	Dump the debug control flow section.
+/-hllsymbols	**h/H**	List the high level language symbols, with address, size and type.
+/-modules	**m/M**	Print a list of modules found in object file.
+/-note	**n/N**	Dump all ELF `.note` sections.
+/-sections	**s/S**	Print a list of sections with start address, length and type.
+/-symbols	**y/Y**	List the low level symbols, with address and length (if known).
	0	Alias for **ACDFHMNSY** (nothing)
	1	Alias for **ACDFhMNSY** (only HLL symbols)
	2	Alias for **ACdFHMNSY** (only section contents)
	3	Alias for **acdfhmnsy** (everything)

Default: **--dump-format=ACdFhmnsy**

Description

With this option you can control which parts of the dump output you want to see.

1. Module list

2. Section list

3. Call graph using the DWARF debug info

4. Data accesses per function

5. Section dump (disassembly)

6. HLL symbol table

7. Assembly level symbol table

8. Note sections

9. Debug control flow section

By default, all parts are dumped, except for parts 3, 4 and 9.

You can limit the number of sections that will be dumped with the options **--sections** and **--section-types**.

Related information

Section 8.6.2, *HLL Dump Output Format*

HLL object dumper option: --expand-symbols (-e)

Command line syntax

--expand-symbols[**=***flag*]**,...**

-e[*flag*]**...**

You can specify one of the following flags:

+/-basic-types	**b/B**	Expand arrays with basic C types.
+/-fullpath	**f/F**	Include the full path to the field level.
+/-gap-info	**g/G**	Insert gap markers where data is not consecutive.
+/-nesting-indicator	**n/N**	Print nesting bars.

Default (no flags): **--expand-symbols=BFGN**

Description

With this option you specify that all struct, union and array symbols are expanded with their fields in the HLL symbol dump.

With **--expand-symbols=+basic-types**, HLL struct and union symbols are listed including all fields. Array members are expanded in one array member per line regardless of the HLL type. For the fields the types and names are indented with 2 spaces.

With **--expand-symbols=+fullpath**, all fields of structs and unions and all members of non-basic type arrays are expanded and prefixed with their parent's names.

With **--expand-symbols=+gap-info**, unused memory in complex data types (structures and unions) between data objects and between code objects is shown as {gap} parts. This option is useful to optimize data memory usage. This option only works if debug information is available in the ELF file.

With **--expand-symbols=+nesting-indicator**, vertical bars (|) are shown to make it easier to see the expanded structs, unions and arrays.

Example

```
hldumptc -F1 hello.elf

---------- HLL symbol table ----------

Address    Size HLL Type            Name
d0000004     20 struct              _dbg_request [dbg.c]
d0000018     80 static char         stdin_buf[80] [_iob.c]

hldumptc -e -F1 hello.elf

---------- HLL symbol table ----------
```

```
Address     Size HLL Type              Name
d0000004     20 struct                _dbg_request [dbg.c]
d0000004      4   int                    _errno
d0000008      1   enum                   nr
d000000c     12   union                  u
d000000c      4     struct                 exit
d000000c      4       int                      status
d000000c      8     struct                 open
d000000c      4       const char         * pathname
d0000010      2       unsigned short int   flags
   ...
d0000018     80 static char            stdin_buf[80] [_iob.c]

hldumptc -eb -F1 hello.elf

---------- HLL symbol table ----------

Address     Size HLL Type              Name
d0000004     20 struct                _dbg_request [dbg.c]
d0000004      4   int                    _errno
d0000008      1   enum                   nr
d000000c     12   union                  u
d000000c      4     struct                 exit
d000000c      4       int                      status
d000000c      8     struct                 open
d000000c      4       const char         * pathname
d0000010      2       unsigned short int   flags
   ...
d0000018     80 static char            stdin_buf[80] [_iob.c]
d0000018      1   char
d0000019      1   char
d000001a      1   char
  ...
d0000067      1   char

hldumptc -ef -F1 hello.elf

---------- HLL symbol table ----------

Address     Size HLL Type              Name
d0000004     20 struct                _dbg_request [dbg.c]
d0000004      4   int                    _dbg_request._errno
d0000008      1   enum                   _dbg_request.nr
d000000c     12   union                  _dbg_request.u
d000000c      4     struct                 _dbg_request.u.exit
d000000c      4       int                    _dbg_request.u.exit.status
d000000c      8     struct                 _dbg_request.u.open
d000000c      4       const char         * _dbg_request.u.open.pathname
d0000010      2       unsigned short int   _dbg_request.u.open.flags
   ...
d0000018     80 static char            stdin_buf[80] [_iob.c]
```

```
hldumptc -eg -F1 hello.elf

---------- HLL symbol table ----------

Address     Size HLL Type             Name
d0000004     20 struct               _dbg_request [dbg.c]
d0000004      4  int                    _errno
d0000008      1  enum                   nr
d0000009      3                         {gap}
d000000c     12  union                  u
d000000c      4    struct                 exit
d000000c      4      int                    status
d0000010      8                           {gap}
d000000c      8    struct                 open
d000000c      4      const char           * pathname
d0000010      2      unsigned short int   flags
d0000012      2                           {gap}
d0000014      4                           {gap}
    ...
d0000018     80 static char           stdin_buf[80] [_iob.c]

hldumptc -en -F1 hello.elf

---------- HLL symbol table ----------

Address     Size HLL Type             Name
d0000004     20 struct               _dbg_request [dbg.c]
d0000004      4 | int                  _errno
d0000008      1 | enum                 nr
d000000c     12 | union                u
d000000c      4 | | struct               exit
d000000c      4 | | | int                  status
d000000c      8 | | struct               open
d000000c      4 | | | const char         * pathname
d0000010      2 | | | unsigned short int  flags
    ...
d0000018     80 static char           stdin_buf[80] [_iob.c]
```

Related information

Section 8.6.2, *HLL Dump Output Format*

HLL object dumper option: --help (-?)

Command line syntax

`--help`

`-?`

Description

Displays an overview of all command line options.

Example

The following invocations all display a list of the available command line options:

```
hldumptc -?
hldumptc --help
hldumptc
```

Related information

-

HLL object dumper option: --hex (-x)

Command line syntax

--hex

-x

Description

With this option you can control the way data sections and code sections are dumped. By default, the contents of data sections are represented by directives. A new directive will be generated for each symbol. ELF labels in the section are used to determine the start of a directive. ROM sections are represented with `.byte`, `.half`, `.word` kind of directives, depending on the size of the data. RAM sections are represented with `.space` directives, with a size operand depending on the data size. This can be either the size specified in the ELF symbol, or the size up to the next label. Code sections are dumped as disassembly.

With option **--hex**, no directives will be generated for ROM data sections and no disassembly dump will be done for code sections. Instead ROM data sections and code sections are dumped as hexadecimal code with ASCII translation. RAM sections will be represented with only a start address and a size indicator.

Example

```
hldumptc -F2 --section=.rodata.hello..2.str hello.elf

---------- Section dump ----------

        .sdecl '.rodata.hello..2.str',DATA AT 0x800007a0
        .sect  '.rodata.hello..2.str'
        .byte 48,65,6c,6c,6f,2c,20,25,73,21,0a,00              ; Hello, %s!..

hldumptc -F2 --section=.rodata.hello..2.str --hex hello.elf

---------- Section dump ----------

                         section 7 (.rodata.hello..2.str):
800007a0 48 65 6c 6c 6f 2c 20 25 73 21 0a 00              Hello, %s!..
```

Related information

Section 8.6.2, *HLL Dump Output Format*

HLL object dumper option: --mcs-section

Command line syntax

--mcs-section=`section`**,...**

Description

By default the section names recognized for the MCS are `.mcstext` and `.mcsdata`.

With this option you can specify additional MCS section names used to recognize MCS sections when disassembling.

Example

```
hldumptc --mcs-section=.mymcstext hello.elf
```

Related information

-

HLL object dumper option: --option-file (-f)

Command line syntax

--option-file=*file*,...

-f *file*,...

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the HLL object dumper.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** multiple times.

Format of an option file

• Multiple arguments on one line in the option file are allowed.

• To include whitespace in an argument, surround the argument with single or double quotes.

• If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

```
"This has a single quote ' embedded"

'This has a double quote " embedded'

'This has a double quote " and a single quote '"' embedded"
```

• When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \
line"

        -> "This is a continuation line"
```

• It is possible to nest command line files up to 25 levels.

Example

Suppose the file myoptions contains the following lines:

```
--symbols=hll
--class=code
hello.elf
```

Specify the option file to the HLL object dumper:

```
hldumptc --option-file=myoptions
```

This is equivalent to the following command line:

```
hldumptc --symbols=hll --class=code hello.elf
```

Related information

-

HLL object dumper option: --output (-o)

Command line syntax

--output=_file_

-o _file_

Description

By default, the HLL object dumper dumps the output on stdout. With this option you specify to dump the information in the specified file.

The default output format is text, but you can specify another output format with option **--output-type**.

Example

hldumptc --output=dump.txt hello.elf

The HLL object dumper dumps the output in file dump.txt.

Related information

HLL object dumper option **--output-type**

HLL object dumper option: --output-type (-T)

Command line syntax

--output-type[**=***type*]

-T[*type*]

You can specify one of the following types:

text	t	Output human readable text.
xml	x	Output XML.

Default: **--output-type=text**

Description

With this option you can specify whether the output is formatted as plain text or as XML.

Related information

HLL object dumper option **--output**

HLL object dumper option: --pcp-section

Command line syntax

`--pcp-section=`*section*`,...`

Description

By default the section names recognized for the PCP are `.pcptext` and `.pcpdata`.

With this option you can specify additional PCP section names used to recognize PCP sections when disassembling.

Example

```
hldumptc --pcp-section=.mypcptext hello.elf
```

Related information

-

HLL object dumper option: --print-mangled-symbols (-P)

Command line syntax

`--print-mangled-symbols`

`-P`

Description

The C++ compiler can generate unreadable symbol names. These symbols cannot easily be related to your C++ source file anymore. Therefore the HLL dumper by default demangles C++ function names and variable names in the HLL symbol table. With this option you can override this default setting and print the mangled names instead.

Example

```
hldumptc hellocpp.elf

---------- HLL symbol table ----------

Address     Size HLL Type              Name
80000e9a     12 void                  __register_finalization_routine()

hldumptc --print-mangled-symbols hellocpp.elf

---------- HLL symbol table ----------

Address     Size HLL Type              Name
80000e9a     12 void                  _Z31__register_finalization_routinev()
```

Related information

-

HLL object dumper option: --sections (-s)

Command line syntax

--sections=name**,...**

-sname**,...**

Description

With this option you can restrict the output to the specified sections only. This option affects the following parts of the output:

Output part	Effect of --sections
Module list	Not restricted
Section list	Only lists the specified sections
Section dump	Only dumps the contents of the specified sections
HLL symbol table	Not restricted
Assembly level symbol table	Only lists symbols defined in the specified sections
Note sections	Not restricted

By default all sections are included.

Related information

Section 8.6.2, *HLL Dump Output Format*

HLL object dumper option: --source-lookup-path (-L)

Command line syntax

`--source-lookup-path=``path`

`-L``path`

Description

With this option you can specify an additional path where your source files are located. If you want to specify multiple paths, use the option **--source-lookup-path** for each separate path.

The order in which the HLL object dumper will search for source files when intermixed disassembly is used, is:

1. The path obtained from the HLL debug information.

2. The path that is specified with the option **--source-lookup-path**. If multiple paths are specified, the paths will be searched for in the order in which they are given on the command line.

Example

Suppose you call the HLL object dumper as follows:

```
hldumptc --disassembly-intermix --source-lookup-path=c:\mylib\src hello.elf
```

First the HLL object dumper looks in the directory found in the HLL debug information of file `hello.elf` for the location of the source file(s). If it does not find the file(s), it looks in the directory `c:\mylib\src`.

Related information

HLL object dumper option **--disassembly-intermix**

HLL object dumper option: --symbols (-S)

Command line syntax

--symbols[**=***type*]

-S[*type*]

You can specify one of the following types:

asm	**a**	Display assembly symbols in code dump.
hll	**h**	Display HLL symbols in code dump.
none	**n**	Display plain addresses in code dump.

Default: **--symbols=asm**

Description

With this option you can control symbolic information in the disassembly and data dump. For data sections this only applies to symbols used as labels at the data addresses. Data within the data sections will never be replaced with symbols.

Only symbols that are available in the ELF or DWARF information are used. If you build an application without HLL debug information the **--symbols=hll** option will result in the same output as with **--symbols=none**. The same applies to the **--symbols=asm** option when all symbols are stripped from the ELF file.

Example

```
hldumptc -F2 hello.elf

----------- Section dump ----------

                                .sdecl '.text._START.libc.reset', CODE AT 0xa0000000
                                .sect  '.text._START.libc.reset'
a0000000 9d 80 48 01  _START: ja      __init_sp

hldumptc --symbols=none -F2 hello.elf

----------- Section dump ----------

                                .sdecl '.text._START.libc.reset', CODE AT 0xa0000000
                                .sect  '.text._START.libc.reset'
a0000000 9d 80 48 01            ja      0x80000290
```

Related information

Section 8.6.2, *HLL Dump Output Format*

HLL object dumper option: --version (-V)

Command line syntax

`--version`

`-V`

Description

Display version information. The HLL object dumper ignores all other options or input files.

Related information

-

HLL object dumper option: --xml-base-filename (-X)

Command line syntax

`--xml-base-filename`

`-X`

Description

With this option the `<File name>` field in the XML output only contains the filename of the object file. By default, any path name, if present, is printed as well.

Example

`hldumptc --output-type=xml --output=hello.xml ../hello.elf`

The field `<File name="../hello.elf">` is used in `hello.xml`.

`hldumptc --output-type=xml --output=hello.xml -X ../hello.elf`

The field `<File name="hello.elf">` is used in `hello.xml`. The path is stripped from the filename.

Related information

HLL object dumper option **--output-type**

10.11. ELF Patch Utility Options

The ELF patch utility **elfpatch** is a tool to change one or more section names or modify data references within a relocatable ELF object file (`.o`) or library (`.a`).

The invocation syntax is:

elfpatch --command-file=`command-file` [`option`]`... ELF-file`

or:

elfpatch --data-reference-modification-file=`file` [`option`]`... ELF-file`

This section describes all options for the ELF patch utility.

For detailed information about the ELF patch utility, see Section 8.7, *ELF Patch Utility*.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

ELF patch utility option: --command-file (-c)

Menu entry

1. Select **Linker » Patching**.

2. Enable the option **Patch object and library files supplied to the linker**.

3. Specify an **ELF patch command file**.

Command line syntax

--command-file=*file*

-c *file*

Description

Read commands from an ELF patch command file (EPF). Such a file contains one or more
rename_section commands and/or **rename_section_by_symbol** commands.

Example

```
elfpatch --command-file=commands.epf mylib.a
```

Related information

Section 8.7, *ELF Patch Utility*

ELF patch utility option: --data-reference-modification-file (-d)

Menu entry

1. Select **Linker » Patching**.

2. Enable the option **Patch object and library files supplied to the linker**.

3. Specify a **Data reference modification file**.

4. From the **Add info section to patched object files** list, select **Full**, **Compact**, **Executed commands** or **None**.

Command line syntax

```
--data-reference-modification-file=file
```

```
-d file
```

Description

Read commands from an ELF patch data reference modification file (DRM). Such a file contains one or more commands to specify a replacement for any access to a specified global variable.

By default the contents of this file is placed in a section called `.VarEd_Info`, unless you specify option **--vared-info-section=no-section**.

Example

To read data reference modifications from file `myproject.drm`, enter:

```
elfpatch --data-reference-modification-file=myproject.drm mylib.a
```

Related information

Section 8.7, *ELF Patch Utility*

ELF patch utility option **--vared-info-section**

ELF patch utility option: --diag

Command line syntax

--diag=[*format*:]{**all** | *msg*[-*msg*],...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas, or you can specify a range.

With this option the ELF patch utility does not patch any files.

Example

To display an explanation of message number 102, enter:

```
elfpatch --diag=102
```

This results in the following message and explanation:

```
F102: cannot delete "<file>"
```

```
The output file or a temporary file could not be deleted. Check if you have
sufficient disk space and if you have write permissions for the specified
file.
```

To write an explanation of all errors and warnings in HTML format to file elfpatch_errors.html, use redirection and enter:

```
elfpatch --diag=html:all > elfpatch_errors.html
```

Related information

-

ELF patch utility option: --help (-?)

Command line syntax

--help[**=***item*]

-?

You can specify the following arguments:

options	**o**	Show extended option descriptions
syntax	**s**	Show a description of the command file syntax

Description

Displays an overview of all command line options. With an argument you can specify which extended information is shown.

When you specify the argument **syntax** you will see a description of the commands you can use in the ELF patch command file and the data reference modification file.

Example

The following invocations all display a list of the available command line options:

```
elfpatch -?
elfpatch --help
elfpatch
```

The following invocation displays the syntax of the command file and data reference modification file:

```
elfpatch --help=syntax
```

```
Syntax of the command file:
rename_section( "<oldname_pattern>", "<newname>" ) // comments
rename_section_by_symbol( "<symbol-name>", "<new-section-name>" ) // comments
Wildcard character * can be used in the oldname_pattern.

Syntax of the data reference modification file:
<function_name>              : <old_variable_name> = <new_variable_name> ; # comments
<global_struct_variable_name> : <old_member_name>   = <new_member_name>   ; # comments
```

Related information

Section 8.7, *ELF Patch Utility*

ELF patch utility option: --no-warnings (-w)

Command line syntax

--no-warnings[**=***number*[-*number*],...]

-w[*number*[-*number*],...]

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.

- If you specify this option but without numbers, all warnings are suppressed.

- If you specify this option with a number or a range, only the specified warnings are suppressed. You can specify the option **--no-warnings=***number* multiple times.

Example

To suppress warnings 753 and 755, enter:

```
elfpatch -c commands.epf --no-warnings=753,755 file.o
```

Related information

ELF patch utility option **--warnings-as-errors** (Treat warnings as errors)

ELF patch utility option: --option-file (-f)

Command line syntax

--option-file=*file*

-f *file*

Description

Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the ELF patch utility.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** (**-f**) multiple times.

If you use '-' instead of a filename it means that the options are read from stdin.

Format of an option file

- Multiple arguments on one line in the option file are allowed.

- To include whitespace in an argument, surround the argument with single or double quotes.

- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

```
"This has a single quote ' embedded"

'This has a double quote " embedded'

'This has a double quote " and a single quote '"' embedded"
```

- When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \
line"

        -> "This is a continuation line"
```

- It is possible to nest command line files up to 25 levels.

Example

Suppose the file myoptions contains the following lines:

```
-c commands.epf
mylib.a
```

Specify the option file to the ELF patch utility:

```
elfpatch --option-file=myoptions
```

This is equivalent to the following command line:

```
elfpatch -c commands.epf mylib.a
```

Related information

-

ELF patch utility option: --output-file (-o)

Command line syntax

--output-file=*file*

-o *file*

Description

By default, the ELF patch utility uses *input-file*.new as the name for the output file. With this option you can specify another name.

Example

elfpatch -c commands.epf --output-file=myfile.o file.o

Related information

-

ELF patch utility option: --report (-r)

Command line syntax

--report=_result_**,...**

-r _result_**,...**

You can specify the following reports:

+/-not-executed-commands	**c/C**	Data reference modification: report commands not executed in the module or the library
+/-global-executed	**e/E**	Global report of all executed commands
+/-not-found-functions	**f/F**	Data reference modification: report functions not found in the module or the library
+/-global-report	**g/G**	Global report of all commands (executed and not executed)
+/-not-found-sections	**s/S**	Section renaming: report sections not found in the module or the entire library
	0	alias for **--report=CEFGS**
	1	alias for **--report=CEFgS**
	2	alias for **--report=CeFGS**
	3	alias for **--report=cefgs**

Default: **--report=CEFGS**

Description

With this option you can report various types of results after processing an ELF file or a library. The reports give detailed information about which section renaming commands (**s**) or which data reference modification commands (**c**) were not executed during the processing of the ELF file or library. You can also list the functions (**f**) that were not found during data reference modification. The global reports (**e** and **g**) give a list of both section renaming commands and data reference modification commands. The reports are issued once after processing an entire library.

Example

To read data reference modifications from file `myproject.drm`, and report the commands that are not executed, enter:

```
elfpatch -dmyproject.drm --report=+not-executed-commands mylib.a
```

Related information

Section 8.7, *ELF Patch Utility*

ELF patch utility option: --vared-info-section

Menu entry

1. Select **Linker » Patching**.

2. Enable the option **Patch object and library files supplied to the linker**.

3. Specify a **Data reference modification file**.

4. From the **Add info section to patched object files** list, select **Full**, **Compact**, **Executed commands** or **None**.

Command line syntax

```
--vared-info-section=flag
```

You can specify the following arguments:

compact	Leave out comments and most white space.
executed-commands	Copy only the executed commands.
full	Copy the entire data reference modification file.
no-section	Do not create the `.VarEd_Info` section.

Default: **full**

Description

With this option you can determine what will be copied from the ELF patch data reference modification file into the new section named `.VarEd_Info`, or you can choose not to generate the `.VarEd_Info` section at all. By default the entire data reference modification file is copied into the `.VarEd_Info` section.

Example

To read data reference modifications from file `myproject.drm`, and create a compact version in a `.VarEd_Info` section, enter:

```
elfpatch -dmyproject.drm --vared-info-section=compact mylib.a
```

To read data reference modifications from file `myproject.drm`, but do not put the contents in a `.VarEd_Info` section, enter:

```
elfpatch -dmyproject.drm --vared-info-section=no-section mylib.a
```

Related information

Section 8.7, *ELF Patch Utility*

ELF patch utility option **--data-reference-modification-file**

ELF patch utility option: --verbose (-v) / -vv

Command line syntax

`--verbose`

`-v / -vv`

Description

With this option you put the ELF patch utility in *verbose* mode. The ELF patch utility prints which rule is executed and how many times a rule is executed. In the *extra verbose* mode, the tool also prints which rule is not executed.

Example

```
elfpatch -c commands.epf --output-file=myfile.o -vv file.o

processing "file.o"
rule at line 2 executed 2 times
rule at line 3 executed 0 times
```

Related information

-

ELF patch utility option: --version (-V)

Command line syntax

`--version`

`-V`

Description

Display version information. The ELF patch utility ignores all other options or input files.

Related information

-

ELF patch utility option: --warnings-as-errors

Command line syntax

```
--warnings-as-errors[=number[-number],...]
```

Description

If the ELF patch utility encounters an error, it stops. When you use this option without arguments, you tell the ELF patch utility to treat all warnings not suppressed by option **--no-warnings** as errors. This means that the exit status of the ELF patch utility will be non-zero after one or more warnings. As a consequence, the ELF patch utility now also stops after encountering a warning.

You can limit this option to specific warnings by specifying a comma-separated list of warning numbers or ranges. In this case, this option takes precedence over option **--no-warnings**.

Related information

ELF patch utility option **--no-warnings** (Suppress some or all warnings)

10.12. Expire Cache Utility Options

With the utility **expiretc** you can limit the size of the cache (C compiler option **--cache**) by removing all files older than a few days or by removing older files until the total size of the cache is smaller than a specified size. See also Section 11.6, *Compiler Cache*.

The invocation syntax is:

expiretc [*option*]... *cache-directory*

The compiler cache is present in the directory ctccache under the specified *cache-directory*.

This section describes all options for the expire cache utility.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Expire cache utility option: --access (-a)

Command line syntax

`--access`

`-a`

Description

Use the last access time instead of the last modification time to determine which files to delete.

Example

expiretc --access --days=7 "*installation-dir*\mproject\.cache"

Related information

-

Expire cache utility option: --days (-d)

Menu entry

1. Select **C/C++ Compiler » Optimization » Compilation Speed**.

2. Enable the option **Cache generated code to improve the compilation speed**.

3. In the **Directory for cached files** field, enter the name for the location of the cache.

    By default this is the .cache directory under your project directory.

4. Specify the **Maximum days files will live in the cache**.

Command line syntax

--days=*n*

-d*n*

Description

Remove all files older than *n* days from the cache.

Example

To remove all files older than seven days, enter:

```
expiretc --days=7 "installation-dir\mproject\.cache"
```

Related information

-

Expire cache utility option: --diag

Command line syntax

--diag=[*format*:]{**all** | *msg*[-*msg*],...}

You can set the following output formats:

html   HTML output.

rtf   Rich Text Format.

text   ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas, or you can specify a range.

With this option the expire cache utility does not remove any files.

Example

To display an explanation of message number 204, enter:

```
expiretc --diag=204
```

This results in the following message and explanation:

```
E204: failed to remove "<file>" <<cause>>

The removal of the indicated file failed. The <cause>
provides more details of the problem.
```

To write an explanation of all errors and warnings in HTML format to file `expiretc_errors.html`, use redirection and enter:

```
expiretc --diag=html:all > expiretc_errors.html
```

Related information

-

Expire cache utility option: --dry-run (-n)

Command line syntax

`--dry-run`

`-n`

Description

With this option you put the expire utility in verbose mode. The utility shows which files would be deleted, without actually removing them.

Related information

Expire cache utility option **--verbose** (Verbose output)

Expire cache utility option: --help (-?)

Command line syntax

--help[**=***item*]

-?

You can specify the following argument:

> **options**   Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

The following invocations all display a list of the available command line options:

```
expiretc -?
expiretc --help
expiretc
```

To see a detailed description of the available options, enter:

```
expiretc --help=options
```

Related information

-

Expire cache utility option: --megabytes (-m)

Menu entry

1. Select **C/C++ Compiler » Optimization » Compilation Speed**.

2. Enable the option **Cache generated code to improve the compilation speed**.

3. In the **Directory for cached files** field, enter the name for the location of the cache.

    By default this is the .cache directory under your project directory.

4. Enable the option **Clear cache upon project clean**.

    *Each time you use **Project » Clean...** the cache is cleared.*

Command line syntax

`--megabytes=`*m*

`-m`*m*

Description

Reduce the size of the cache to *m* MBytes by removing files from the cache, starting with the oldest file. With a size of 0 (zero) you clear the entire cache.

Example

To reduce the compiler cache size to 4 MB, enter:

```
expiretc --megabytes=4 "installation-dir\mproject\.cache"
```

Older files are removed until the total size of the cache is smaller than 4 MB.

To clear the compiler cache, enter:

```
expiretc --megabytes=0 "installation-dir\mproject\.cache"
```

Related information

-

Expire cache utility option: --totals (-t)

Command line syntax

--totals

-t

Description

Show the total size of the cache and the number of directories and files. This option is implicit when invoked without the **--days** and **--megabytes** options.

Example

```
expiretc -t "installation-dir\mproject\.cache"

installation-dir\mproject\.cache\ctccache:
1 MB, 3 directories, 3 files
```

Related information

-

Expire cache utility option: --verbose (-v)

Command line syntax

`--verbose`

`-v`

Description

With this option you put the expire cache utility in verbose mode. The utility shows which files are being deleted.

Example

```
expiretc -v --megabytes=0 "installation-dir\mproject\.cache"

2014-07-03 12:36:15 installation-dir\mproject\.cache\ctccache\cstart\30aa7935
2014-07-03 12:36:17 installation-dir\mproject\.cache\ctccache\myproject\6f0a3ba4
2014-07-03 12:36:18 installation-dir\mproject\.cache\ctccache\sync_on_halt\bab2c716
```

Related information

-

Expire cache utility option: --version (-V)

Command line syntax

`--version`

`-V`

Description

Display version information and exit. The expire cache utility ignores all other options.

Related information

-

10.13. Integrity Check Utility Options

The Integrity Check utility **ichk** is a tool to check the integrity of an application. It checks access restrictions on memory in light of ASIL specifications and MPU configuration.

The invocation syntax is:

ichk [*option*]... *ELF_file*

This section describes all options for the Integrity Check utility.

For detailed information about the Integrity Check utility, see Section 8.10, *Integrity Check Utility*.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+**_longflag_. To switch a flag off, use an uppercase letter or a **-**_longflag_. Separate _longflags_ with commas. The following two invocations are equivalent:

```
ichk -ca test.elf
ichk --check=+access-rights test.elf
```

When you do not specify an option, a default value may become active.

Integrity check utility option: --check

Command line syntax

`--check=`*flags*

You can set the following flags:

> **+/-access-rights**   **a/A**   check function and variables access rights

Default: `--check=a`

Description

This option controls a number of checks to detect errors during program execution.

With **--check=+access-rights**, the Integrity Check utility checks the ELF file for possible violations of safety restrictions.

Related information

-

Integrity check utility option: --diag

Command line syntax

`--diag=`[*format*:]{**all** | *msg*[*-msg*],...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. The Integrity Check utility does not process any files. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas, or you can specify a range.

Example

To display an explanation of message number 281, enter:

```
ichk --diag=281
```

This results in the following message and explanation:

```
E281: safety area starting at address <address> overlaps with other area
```

```
The current safety area overlaps with an earlier specified safety area.
```

To write an explanation of all errors and warnings in HTML format to file `ichkerrors.html`, use redirection and enter:

```
ichk --diag=html:all > ichkerrors.html
```

Related information

-

Integrity check utility option: --error-file

Command line syntax

--error-file[**=***file*]

Description

With this option the integrity check utility redirects error messages to a file. If you do not specify a filename, the error file will be named after the output file with extension `.eri`.

Example

To write errors to `ichkerrors.eri` instead of `stderr`, enter:

```
ichk --error-file=ichkerrors.eri test.elf
```

Related information

-

Integrity check utility option: --help (-?)

Command line syntax

--help[**=***item*]

-?

You can specify the following arguments:

options **o** Show extended option descriptions

Description

Displays an overview of all command line options.

Example

The following invocations all display a list of the available command line options:

```
ichk -?
ichk --help
ichk
```

Related information

-

Integrity check utility option: --option-file (-f)

Command line syntax

--option-file=_file_,...

-f _file_,...

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the Integrity Check utility.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** multiple times.

Format of an option file

* Multiple arguments on one line in the option file are allowed.

* To include whitespace in an argument, surround the argument with single or double quotes.

* If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

  ```
  "This has a single quote ' embedded"

  'This has a double quote " embedded'

  'This has a double quote " and a single quote '"' embedded"
  ```

* When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

  ```
  "This is a continuation \
  line"

            -> "This is a continuation line"
  ```

* It is possible to nest command line files up to 25 levels.

Example

Suppose the file myoptions contains the following lines:

```
--check=+access-rights
--output=ichkhello.txt
hello.elf
```

Specify the option file to the integrity check utility:

```
ichk --option-file=myoptions
```

This is equivalent to the following command line:

```
ichk --check=+access-rights --output=ichkhello.txt hello.elf
```

Related information

-

Integrity check utility option: --output (-o)

Command line syntax

--output=*file*

-o *file*

Description

By default, the Integrity Check utility writes the output on stdout. With this option you specify to write the information to the specified file. When an error occurs, the generated output file will not be removed.

Example

```
ichk --output=ichkhello.txt hello.elf
```

The Integrity Check utility writes the output to file ichkhello.txt.

Related information

-

Integrity check utility option: --version (-V)

Command line syntax

`--version`

`-V`

Description

Display version information. The Integrity Check utility ignores all other options or input files.

Related information

-

Chapter 11. Influencing the Build Time

In general many settings have influence on the build time of a project. Any change in the tool settings of your project source will have more or less impact on the build time. The following sections describe several issues that can have significant influence on the build time.

11.1. SFR File

SFR files can define such a large number of SFRs that compiling the SFR file alone already takes up a significant part of the build time. To reduce the build time:

* By default, the tools do not automatically include the SFR file. You should include the SFR file only in the source modules where the SFRs are used, with a `#include` directive. In Eclipse make sure that the automatic inclusion option is disabled. You can find this option on the "**C/C++ Compiler »Preprocessing**" and the "**Assembler » Preprocessing**" pages.

    When you include the SFR file in the source, be aware that the SFR files are in the `sfr` subdirectory of the include files, so you must use: `#include <sfr/regtc1796b.sfr>`

11.2. MIL Linking

With MIL linking (see Section 4.6.1, *Generic Optimizations (frontend)*) it is possible to let the compiler apply optimizations application wide. This can yield significant optimization improvements, but the build times can also be significantly longer. MIL linking itself can require significant time, but also the changed build process implies longer build times. The MIL linking settings in Eclipse are:

* **Build for application wide optimizations (MIL linking)**

    This enables MIL linking. The build process changes: the C files are translated to intermediate code (MIL files) and the generated MIL files of the whole project are linked together by the C compiler. The next step depends on the setting of the option below.

* **Application wide optimization mode: Optimize more/Build slower**

    When this option is enabled, the compiler runs the code generator immediately on the completely linked MIL stream, which represents the entire application. This way the code generator can perform several optimizations, such as "code compaction", at application scope. But this also requires significantly more memory and requires more time to generate code. Besides that, it is no longer possible to do incremental builds. With each build the full MIL linking phase and code generation has to be done, even with the smallest change that would in a normal build (not MIL linking) require only a single module to be translated.

* **Application wide optimization mode: Optimize less/Build faster**

    When this option is disabled, the compiler splits the MIL stream after MIL linking in separate modules. This allows the code generation to be performed for the modified modules only, and will therefore be faster than with the other option enabled. Although the MIL stream is split in separate modules after MIL linking, it still may happen that modifying a single C source file results in multiple MIL files to be

compiled. This is a natural result of global optimizations, where the code generated for multiple modules was affected by the change.

In general, if you do not need code compaction, for example because you are optimizing fully for speed, it is recommended to choose **Optimize less/Build faster**.

11.3. Optimization Options

In general any optimization may require more work to be done by the compiler. But this does not mean that disabling all optimizations (level 0) gives the fastest compilation time. Disabling optimizations may result in more code being generated, resulting in more work for other parts of the compiler, like for example the register allocator.

11.4. Automatic Inlining

Automatic inlining is an optimization which can result in significant longer build time. The overall functions will get bigger, often making it possible to do more optimizations. But also often resulting in more registers to be in use in a function, giving the register allocation a tougher job.

11.5. Code Compaction

When you disable the code compaction optimization, the build times may be shorter. Certainly when MIL linking is used where the full application is passed as a single MIL stream to the code generation. Code compaction is however an optimization which can make a huge difference when optimizing for code size. When size matters it makes no sense to disable this option. When you choose to optimize for speed (**--tradeoff=0**) the code compaction is automatically disabled.

11.6. Compiler Cache

The C compiler has support for caching intermediate results to avoid full compilations. When the source code after preprocessing and relevant compiler options and the compiler version are the same as in a previous invocation, the previous result is copied to the output file. The cache only works when there is a single C input file and a single output file (no **--mil-split**).

To enable caching from Eclipse:

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » Optimization » Compilation Speed**.

4. Enable the option **Cache generated code to improve the compilation speed**.

5. In the **Directory for cached files** field, enter the name for the location of the cache.

    By default this is the .cache directory under your project directory.

6. Specify the **Maximum days files will live in the cache**.

7. (Optional) Enable the option **Clear cache upon project clean**.

    *Each time you use **Project » Clean...** the cache is cleared.*

Eclipse calls the C compiler with option **--cache**. The cache directory may be shared, for instance by placing it on a network drive. The compiler creates a directory `ctccache` in the specified directory.

When a result from the cache is used, the C compiler generates a comment line in the assembly source file to notify that. In that case be aware of the following:

*   In case source merging is enabled an older version of the source is still shown. As long as a source change has no effect on the preprocessed code, the cached version of the output file is used.

*   Some options, like **--define**, **--include-directory** and **--output** are not part of the hash used for the cache. As long as a change in these options has no influence on the preprocessed code, the cached version of the output is used. This means that the options listed as comments in the generated assembly file might not match the options actually used.

With every compilation of a file that results in a cache miss, a new file is stored in the cache. Old files are not removed from the cache automatically because that would slow down the compiler too much. To keep the cache size reasonable specify a maximum number of days the files will live in the cache. Eclipse uses the utility **expiretc** for this. It is recommended to run this utility frequently, for example with each time the project is linked. For more information on this utility see Section 8.8, *Expire Cache Utility*.

11.7. Header Files

Many applications include all header files in each module, often by including them all within a single include file. Processing header files takes time. It is a good programming practice to only include the header files that are really required in a module, because:

*   it is clear what interfaces are used by a module

*   an incremental build after modifying a header file results in less modules required to be rebuild

*   it reduces compile time

11.8. Parallel Build

The make utility **amk**, which is used by Eclipse, has a feature to build jobs in parallel. This means that multiple modules can be compiled in parallel. With today's multi-core processors this means that each core can be fully utilized. In practice even on single core machines the compile time decreases when using parallel jobs. On multi-core machines the build time even improves further when specifying more parallel jobs than the number of cores.

In Eclipse you can control the parallel build behavior:

1. From the **Project** menu, select **Properties for**

    The Properties dialog appears.

2. In the left pane, select **C/C++ Build**.

    In the right pane the C/C++ Build page appears.

3. On the Behaviour tab, select **Use parallel build**.

4. You can specify the number of parallel jobs, or you can use an optimal number of jobs. In the last case, **amk** will fork as many jobs in parallel as cores are available.

11.9. Section Concatenation

By default the linker does not merge sections with the same name into one section. With the assembler option **--concatenate-sections** the assembler uses the section attribute concat, instructing the linker to merge sections with the same name.

The advantage of section concatenation is faster locating, because there are less sections to locate.

The disadvantage of section concatenation is less efficient memory use, because of alignment gaps between (sequentially concatenated) sections.

Chapter 12. Profiling

Profiling is the process of collecting statistical data about a running application. With these data you can analyze which functions are called, how often they are called and what their execution time is. This chapter describes the TASKING profiling method with code instrumentation techniques and static profiling.

12.1. What is Profiling?

Profiling is a collection of methods to gather data about your application which helps you to identify code fragments where execution consumes the greatest amount of time.

TASKING supplies a number of profiler tools each dedicated to solve a particular type of performance tuning problem. Performance problems can be solved by:

• Identifying time-consuming algorithms and rewrite the code using a more time-efficient algorithm.

• Identifying time-consuming functions and select the appropriate compiler optimizations for these functions (for example, enable loop unrolling or function inlining).

• Identifying time consuming loops and add the appropriate pragmas to enable the compiler to further optimize these loops.

A profiler helps you to find and identify the time consuming constructs and provides you this way with valuable information to optimize your application.

TASKING employs various schemes for collecting profiling data, depending on the capabilities of the target system and different information needs.

12.1.1. Methods of Profiling

There are several methods of profiling: recording by an instruction set simulator, profiling with code instrumentation techniques (dynamic profiling) and profiling by the C compiler at compile time (static profiling). Each method has its advantages and disadvantages.

Profiling by an instruction set simulator

One way to gather profiling information is built into the instruction set simulator (ISS). The ISS records the time consumed by each instruction that is executed. The debugger then retrieves this information and correlates the time spent for individual instructions to C source statements.

Advantages

• it gives (cycle) accurate information with extreme fine granularity

• the executed code is identical to the non-profiled code

Disadvantages

• the method requires an ISS as execution environment

Profiling using code instrumentation techniques (Dynamic Profiling)

The TASKING C compiler has an option to add code to your application which takes care of the profiling process. This is called code instrumentation. The gathered profiling data is first stored in the target's memory and will be written to a file when the application finishes execution or when the function `__prof_cleanup()` is called.

Advantages

- it can give a complete call graph of the application annotated with the time spent in each function and basic block

- this profiling method is execution environment independent

- the application is profiled while it executes on its aimed target taking real-life input

Disadvantage

- instrumentation code creates a significant run-time overhead, and instrumentation code and gathered data take up target memory

This method provides a valuable complement to the other two methods and is described into more detail below.

Profiling estimation by the C compiler (Static Profiling)

The TASKING C compiler has an option to generate static profile information through various heuristics and estimates. The profiling data produced this way at compile time is stored in an XML file, which can be processed and displayed using the same tools used for dynamic (run-time) profiling.

Advantages

- it can give a give a quick estimation of the time spent in each function and basic block

- this profiling method is execution environment independent

- the application is profiled at compile time

- it requires no extra code instrumentation, so no extra run-time overhead

Disadvantage

- it is an estimation by the compiler and therefore less accurate than dynamic profiling

This method also is described into more detail below.

12.2. Profiling using Code Instrumentation (Dynamic Profiling)

Profiling can be used to determine which parts of a program take most of the execution time.

Once the collected data are presented, it may reveal which pieces of your code execute slower than expected and which functions contribute most to the overall execution time of a program. It gives you

also information about which functions are called more or less often than expected. This information not only reveal design flaws or bugs that had otherwise been unnoticed, it also reveals parts of the program which can be effectively optimized.

Important considerations

The dynamic profiling method adds code to your original application which is needed to gather the profiling data. Therefore, the code size of your application increases. Furthermore, during the profiling process, the gathered data is initially stored into dynamically allocated memory of the target. The heap of your application should be large enough to store this data. Since code instrumentation is done by the compiler, assembly functions used in your program do not show up in the profile.

The profiling information is collected during the actual execution of the program. Therefore, the input of the program influences the results. If a part/function of the program is not activated while the program is profiled, no profile data is generated for that part/function.

When you use dynamic profiling on a target board, function `clock()` must be added to the application to get profiling timer results.

It is *not* possible to profile applications that are compiled with the optimization code compaction (C compiler option **--optimize=+compact**). Therefore, when you turn profiling on, the compiler automatically disables parts of the code compaction optimization.

Overview of steps to perform

To obtain a profile using code instrumentation, perform the following steps:

1. Compile and link your program with profiling enabled

2. Execute the program to generate the profile data

3. Display the profile

First you need a completed project. If you are not using your own project, use the `tc_profiling` example as described below.

1. From the **File** menu, select **Import...**

    The Import dialog appears.

2. Select **TASKING C/C++ » TASKING TriCore Example Projects** and click **Next**.

3. In the **Example projects** box, disable all projects except `tc_profiling` .

4. Click **Finish**.

    The `tc_profiling` project should now be visible in the C/C++ view.

12.2.1. Step 1: Build your Application for Profiling

The first step is to add the code that takes care of the profiling, to your application. This is done with C compiler options:

1. From the **Project** menu, select **Properties for**

   The Properties for profiling dialog box appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   In the right pane the Settings appear.

3. On the Tool Settings tab, expand the **C/C++ Compiler** entry and select **Debugging**.

4. Enable one or more of the following **Generate profiling information** options (the sample `tc_profiling` project already has profiling options enabled).

   • **for block counters** (not in combination with Call graph or Function timers)

   • **to build a call graph** (not in combination with Block counters)

   • **for function counters**

   • **for function timers** (not in combination with Block counters/Function counters)

   > Note that the more detailed information you request, the larger the overhead in terms of execution time, code size and heap space needed. The option Generate symbolic debug information (**--debug**) does not affect profiling, execution time or code size.

   Block counters (not in combination with Call graph or Function timers)

   This will instrument the code to perform basic block counting. As the program runs, it will count how many time it executed each branch of each if statement, each iteration of a for loop, and so on. Note that though you can combine Block counters with Function counters, this has no effect because Function counters is only a subset of Block counters.

   Call graph (not in combination with Block counters)

   This will instrument the code to reconstruct the run-time call graph. As the program runs it associates the caller with the gathered profiling data.

   Function counters

   This will instrument the code to perform function call counting. This is a subset of the basic Block counters.

   Function timers (not in combination with Block counters/Function counters)

   This will instrument the code to measure the time spent in a function. This includes the time spent in all called functions (callees). Also the time spent in interrupt functions is taken into account, unless you enable option **Exclude time spent in interrupt functions**.

   For the command line, see the C compiler option **--profile** (**-p**).

   Profiling is only possible with optimization levels 0, 1 and 2. So:

5. Open the **Optimization** page and set the **Optimization level** to **2 - Optimize more**.

6. Click **OK** to apply the new option settings and rebuild the project (🐞).

12.2.1.1. Profiling Modules and C Libraries

Profiling individual modules

It is possible to profile individual C modules. In this case only limited profiling data is gathered for the functions in the modules compiled without the profiling option. When you use the suboption **Call graph**, the profiling data reveals which profiled functions are called by non-profiled functions. The profiling data does not show how often and from where the non-profiled functions themselves are called. Though this does not affect the flat profile, it might reduce the usefulness of the call graph.

Profiling C library functions

Eclipse and/or the control program will link your program with the standard version of the C library. Functions from this library which are used in your application, will not be profiled. If you do want to incorporate the library functions in the profile, you must set the appropriate C compiler options in the C library makefiles and rebuild the library.

12.2.1.2. Linking Profiling Libraries

When building your application, the application must be linked against the corresponding profile library. Eclipse (or the control program) automatically select the correct library based on the profiling options you specified. However, if you compile, assemble and link your application manually, make sure you specify the correct library.

See Section 7.3, *Linking with Libraries* for an overview of the (profiling) libraries.

12.2.2. Step 2: Execute the Application

Once you have compiled and linked the application for profiling, it must be executed to generate the profiling data. Run the program as usual: the program should run normally taking the same input as usual and producing the same output as usual. The application will run somewhat slower than normal because of the extra time spent on collecting the profiling data.

Follow the steps below to run the application on the TASKING simulator, using the debugger. (In fact, you can run the application also on a target board.)

1. From the **Debug** menu, select **Debug Configurations...**

   The Debug Configurations dialog appears.

2. Select **TASKING C/C++ Debugger** and click the **New launch configuration** button (🗋) to add a new configuration.

3. In the **Name** field enter the name of the configuration. By default, this is the name of the project, but you can give your configuration any name you want to distinguish it from the project name. For example enter tc_profiling.simulator to identify the simulator debug configuration.

4.   On the **Target** tab, select the **TriCore 1.6.X Instruction Set Simulator - Production Device**.

5.   Click the **Apply** button.

6.   Click the **Debug** button to start the debugger and launch the profiling application.

     Eclipse will open the TASKING Debug perspective (as specified in the configuration) and asks for confirmation.

7.   Click **Yes** to open the TASKING Debug perspective.

     The TASKING Debug perspective opens while the application has stopped before it enters main()

8.   From the main toolbar, click on the ▯▶ (Resume) button.

     A file system simulation (FSS) view appears in which the application outputs the results.

     When the program has finished, the collected profiling data is saved (for details see 'After execution' below).

Startup code

The startup code initializes the profiling functions by calling the function `__prof_init()`. Eclipse will automatically make the required modifications to the startup code. Or, when you use the control program, this extracts the correct startup code from the C library.

If you use your own startup code, you must manually insert a call to the function `__prof_init` just before the call to `main` and its stack setup.

An application can have multiple entry points, such as `main()` and other functions that are called by interrupt. This does not affect the profiling process.

Small heap problem

When the program does not run as usual, this is typically caused by a shortage of heap space. In this case a message is issued (when running with file system simulation, it is displayed on the Debug console). To solve this problem, increase the size of the heap. Information about the heap is stored in the linker script file (.lsl) file which is automatically added when a project is created.

1.   From the **Project** menu, select **Properties for**

     The Properties dialog appears.

2.   In the left pane, expand **C/C++ Build** and select **Stack/Heap**.

     In the right pane the Stack/Heap property page appears.

3.   Enter larger values for **heap** and click **OK**.

     The project LSL file is updated automatically according to the new settings.

Presumable incorrect call graph

The call graph is based on the *compiled* source code. Due to compiler optimizations the call graph may therefor seem incorrect at first sight. For example, the compiler can replace a function call immediately followed by a return instruction by a jump to the callee, thereby merging the callee function with the caller function. In this case the time spent in the callee function is not recorded separately anymore, but added to the time spent in the caller function (which, as said before, now holds the callee function). This represents exactly the structure of your source in assembly but may differ from the structure in the initial C source.

After execution

When the program has finished (returning from `main()`), the exit code calls the function `__prof_cleanup(void)`. This function writes the gathered profiling data to a file on the host system using the debugger's file system simulation features. If your program does *not* return from `main()`, you can force this by inserting a call to the function `__prof_cleanup()` in your application source code. Please note the double underscores when calling from C code!

The resulting profiling data file is named `amon.prf`.

> If your program does not run under control of the debugger and therefore cannot use the file system simulation (FSS) functionality to write a file to the host system, you must implement a way to pass the profiling data gathered on the target to the host. Adapt the function `__prof_cleanup()` in the profiling libraries or the underlying I/O functions for this purpose.

12.2.3. Step 3: Displaying Profiling Results

After the function `__prof_cleanup()` has been executed, the result of the profiler can be displayed in the TASKING Profiler perspective. The profiling data in the file `amon.prf` is then converted to an XML file. This file is read and its information is displayed. To view the profiling information, open the TASKING Profiler perspective:

1. From the **Window** menu, select **Open Perspective** » **Other...**

    The Open Perspective dialog appears.

2. Select the **TASKING Profiler** perspective and click **OK**.

    The TASKING Profiler perspective opens.
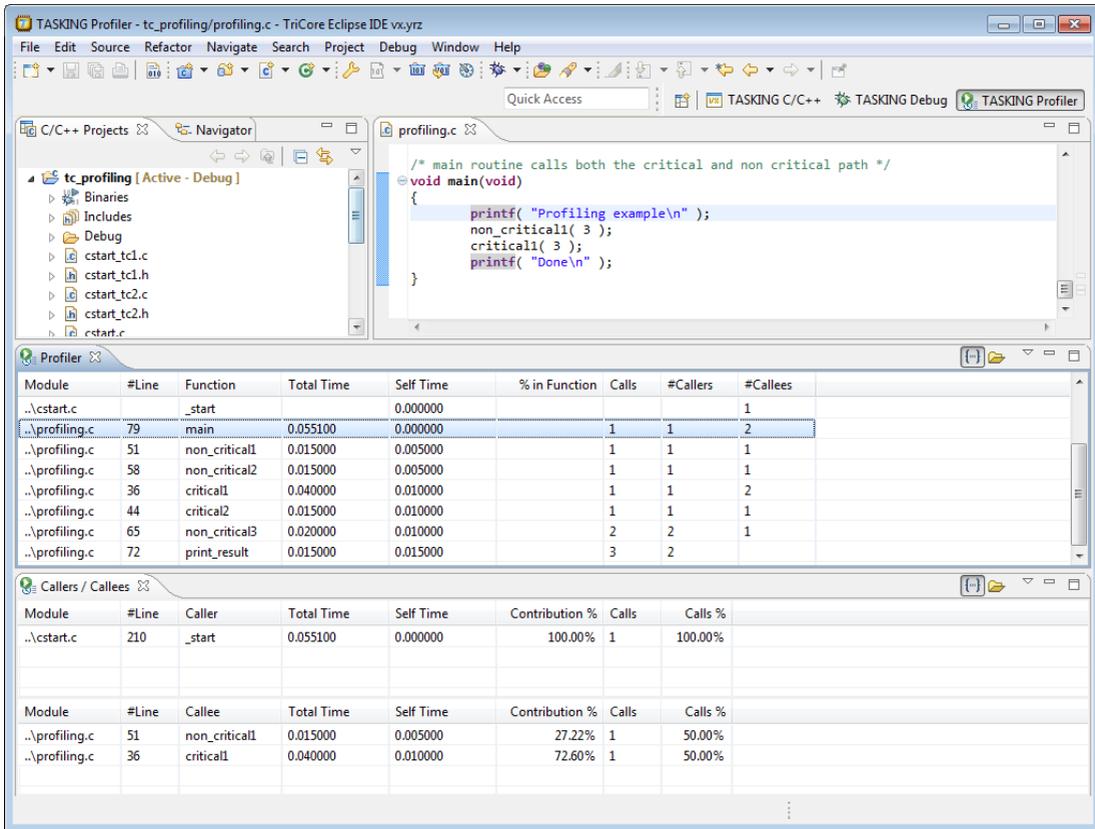
The TASKING Profiler perspective

The TASKING Profiler perspective contains the following Views:

Profiler view	Shows the profiling information of all functions in all C source modules belonging to your application.
Callers / Callees view	The first table in this view, the *callers* table, shows the functions that called the focus function.
	The second table in this view, the *callees* table, shows the functions that are called by the focus function.

- Clicking on a function (or on its table row) makes it the focus function.

- Double-clicking on a function, opens the appropriate C source module in the Editor view at the location of the function definition.

- To sort the rows in the table, click on one of the column headers.

The profiling information

Based on the profiling options you have set before compiling your application, some profiling data may be present and some may be not. The columns in the tables represent the following information:

Module	The C source module in which the function resides.
#Line	The line number of the function definition in the C source module.
Function	The function for which profiling data is gathered and (if present) the code blocks in each function. To show or hide the block counts, in the Profiler view click the **Menu** button (▽) and select **Show Block Counts**.
Total Time	The total amount of time in seconds that was spent in this function and all of its sub-functions.
Self Time	The amount of time in seconds that was spent in the function itself. This excludes the time spent in the sub-functions. So, self time = function's total time - total times of the called functions.
% in Function	This is the relative amount of time spent in this function, calculated as a percentage of the total application time. These should add up to 100%. The total application time is determined by taking the total time of the call graph. This is usually main. Example:

```
Total time of main: 0.002000
Self time of function foo:  0.000100
%in Function = (0.000100 / 0.002000) * 100 =  5%
```

Calls	Number of times the function has been executed.
#Callers	Number of functions by which the function was called.
#Callees	Number of functions that was actually called from this function.
Contribution %	In the caller table: shows for which part (in percent) the caller contributes to the time spent in the focus function. In the callee table: shows how much time the focus function has spent relatively in each of its callees.
Calls %	In the caller table: shows how often each callee was called as a percentage of all calls from the focus function. In the callee table: shows how often the focus function was called from a particular caller as a percentage of all calls to the focus function.

Common toolbar icons

Icon	Action	Description
{ⁱⁱ}	Show/Hide Block Counts	Toggle. If enabled, shows profiling information for block counters.
📂	Select Profiling File(s)	Opens a dialog where you can specify profiling files for display.

Viewing previously recorded profiling results, combining results

Each time you run your application, new profiling information is gathered and stored in the file `amon.prf`. You can store previous results by renaming the file `amon.prf` (keep the extension `.prf`); this prevents the existing `amon.prf` from being overwritten by the new `amon.prf`. At any time, you can reload these profiling results in the profiler. You can even load multiple `.prf` files into the Profiler to view the combined results.

First, open the TASKING Profiler perspective if it is not open anymore:

1. In the Profiler view, click on the 📂 (Select Profiling File(s)) button.

    The Select Profiling File(s) dialog appears.

2. In the **Profiling Type** group box, select **Dynamic Profiling**.

3. In the **Profiling Files** group box, disable the option **Use default**.

4. Click the **Add...** button, select the `.prf` files you want to load and click **Open** to confirm your choice.

5. Make sure the correct symbol file is selected, in this example `tc_profiling.elf`.

6. Click **OK** to finish.

12.3. Profiling at Compile Time (Static Profiling)

Just as with dynamic profiling, static profiling can be used to determine which parts of a program take most of the execution time. It can provide a good alternative if you do not want that your code is affected by extra code.

Overview of steps to perform

To obtain a profile using code instrumentation, perform the following steps:

1. Compile and link your program with static profiling enabled

2. Display the profile

First you need a completed project. If you are not using your own project, use the `tc_profiling` example as described in Section 12.2, *Profiling using Code Instrumentation (Dynamic Profiling)*.

12.3.1. Step 1: Build your Application with Static Profiling

The first step is to tell the C compiler to make an estimation of the profiling information of your application. This is done with C compiler options:

1. From the **Project** menu, select **Properties for**

    The Properties for profiling dialog box appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

In the right pane the Settings appear.

3. On the Tool Settings tab, expand the **C/C++ Compiler** entry and select **Debugging**.

4. Enable **Static profiling**.

   For the command line, see the C compiler option **--profile** (**-p**).

   Profiling is only possible with optimization levels 0, 1 and 2. So:

5. Open the **Optimization** page and set the **Optimization level** to **2 - Optimize more**.

6. Click **OK** to apply the new option settings and rebuild the project (![icon]).

12.3.2. Step 2: Displaying Static Profiling Results

After your project has been built with static profiling, the result of the profiler can be displayed in the TASKING Profiler perspective. The profiling data of each individual file (.sxml), is combined in the XML file tc_profiling.xprof. This file is read and its information is displayed. To view the profiling information, open the TASKING Profiler perspective:

1. From the **Window** menu, select **Open Perspective** » **Other...**

   The Open Perspective dialog appears.

2. Select the **TASKING Profiler** perspective and click **OK**.

   The TASKING Profiler perspective opens. This perspective is explained in Section 12.2.3, Step 3: Displaying Profiling Results

To display static profiling information in the Profiler view

1. In the Profiler view, click on the 📂 (Select Profiling File(s)) button.

   The Select Profiling File(s) dialog appears.

2. In the **Profiling Type** group box, select **Static Profiling**.

3. In the **Static Profiling File** group box, enable the option **Use default**.

   By default, the file *project*.xprof is used (tc_profiling.xprof). If you want to specify another file, disable the option **Use default** and use the edit field and/or Browse button to specify a static profiling file (.xprof).

4. Click **OK** to finish.

Chapter 13. Libraries

This chapter contains an overview of all library functions that you can call in your C source. This includes all functions of the standard C library (ISO C99) and some functions of the floating-point library.

Section 13.1, *Library Functions*, gives an overview of all library functions you can use, grouped per header file. A number of functions declared in `wchar.h` are parallel to functions in other header files. These are discussed together.

Section 13.2, *C Library Reentrancy*, gives an overview of which functions are reentrant and which are not.

C library / floating-point library / run-time library

The following libraries are included in the TriCore toolset. Both Eclipse and the control program **cctc** automatically select the appropriate libraries depending on the specified options.

Libraries	Description
libc[s].a libc[s]_fpu.a	C libraries Optional letter: s = single precision floating-point (control program option **--fp-model=+float**) _fpu = with FPU instructions (default, control program option **--fp-model=-soft**)
libfp[t].a libfp[t]_fpu.a	Floating-point libraries (contains floating-point functions needed by the C compiler) Optional letter: t = trapping (control program option **--fp-model=+trap**) _fpu = with FPU instructions (default, control program option **--fp-model=-soft**)
librt.a	Run-time library (contains other run-time functions needed by the C compiler)
libpb.a libpc.a libpct.a libpd.a libpt.a	Profiling libraries pb = block/function counter pc = call graph pct = call graph and timing pd = dummy pt = function timing

For the C libraries and profiling libraries also MIL library variants are present (files with extension `.ma`).

Sources for the libraries are present in the directories `lib\src, lib\src.*` in the form of an executable. If you run the executable it will extract the sources in the corresponding directory.

Floating-point library with trapping

If you use the trapping floating-point library (`libfpt.a` or `libfpt_fpu.a`), exceptional floating-point cases are intercepted and can be handled separately by an application defined trap handler. Using this

library decreases the execution speed of your application. The header file `except.h` contains the TriCore specific software floating-point trap handling interface definition. See Section 13.1.7, *except.h* for the interface functions and a list of floating-point exceptions, such as overflow and underflow.

See the `fptrap` example delivered with the product for an example of how floating-point exceptions can be handled. The file `fptrap.c` installs a floating-point trap handler which is called when a floating-point exception occurs. In this example some of the possible exceptions are generated. You can use this example as a starting point to write your own program which handles the exception without "hanging" the program or producing incorrect output. This handler overrules the default handler (`exception_handler`) delivered in the floating-point run-time library in `except.c`. Note that the default handler is not reentrant.

Library with FPU instructions

The TASKING VX-toolset for TriCore contains special variants of the C libraries and floating-point libraries that use floating-point unit (FPU) instructions (`libc_fpu.a` (double-precision), `libcs_fpu.a` (single-precision), `libfp_fpu.a` and `libfpt_fpu.a`). By default, the C compiler assumes an FPU is present for the selected processor. This means that the C compiler can generate hardware floating-point instructions. Sometimes special constructs are necessary, for example for optimizations, where the FPU has no standard instructions for. The FPU variant of the library has functions that implement such a special construct, for example to compare if two floats are equal, that uses extra code around standard FPU instructions. In that case the C compiler calls the function in the FPU variant of the floating-point library.

For example, with

```
int fisnequal( float a, float b )
{
 return ( a != b )
}
```

the C compiler generates a normal FPU instruction:

```
    cmp.f  d2,d4,d5
    ne     d2,d2,#2
```

But for example, with

```
int fisequal( float a, float b )
{
 return ( a == b )
}
```

the C compiler makes a call to the `__f_feq` function in the FPU variant of the floating-point library, because there is no standard FPU instruction for this construct:

```
    j        __f_feq
    .extern  __f_feq
```

The call to `__f_feq` is implemented as (see function `equals_spf` in `lib\src.fp\equ_spf_fpu.c`):

```
__asm(
        "cmp.f   %0,%2,%3    \n"
    "    mfcr    %1,#0xFE04  \n"
```

```
"       and     %0,%0,#0x2  \n"
"       ne      %0,%0,#0x0  \n"
:       "=&d"(result), "=d"(PSW)
:       "d"(fop1), "d"(fop2)
);
```

Note however, that the C compiler might not generate a call to the FPU library function if you are optimizing for speed (**--tradeoff=0**).

Eclipse and the control program automatically select the correct library based on the selected processor. If you do not want to use FPU instructions and FPU constructs in the FPU variants of the library, you can specify control program option **--fp-model=+soft**. In this case software floating-point is used and the libraries are selected which are built with this same **--fp-model=+soft** option (without the `_fpu` addition).

C++ library

The TASKING C++ compiler supports the STLport C++ libraries. STLport is a multi-platform ISO C++ Standard Library implementation. It is a free, open-source product, which is delivered with the TASKING C++ compiler. The library supports standard templates and I/O streams. The libraries have been built with C++ compiler option **--c++11**.

I/O streams are disabled by default. So, when you use standard I/O streams in your C++ source, e.g. `cout`, make sure to specify C++ compiler option **--io-streams**.

The include files for the STLport C++ libraries are present in directory `include.stl` relative to the product installation directory.

You can find more information on the STLport library on the following site: http://stlport.sourceforge.net/

Also read the license agreement on http://stlport.sourceforge.net/License.shtml. This license agreement is applicable to the STLport C++ library only. All other product components fall under the TASKING license agreement.

For an STL Programmer's Guide you can see http://www.sgi.com/tech/stl/index.html

The following C++ libraries are delivered with the product:

Libraries	Description
libcp[s][x].a libcp[s][x]_fpu.a	C++ libraries Optional letter: s = single precision floating-point x = exception handling _fpu = with FPU instructions (default, control program option **--fp-model=-soft**)
libstl[s]x.a libstl[s]x_fpu.a	STLport C++ libraries (exception handling variants only) Optional letter: s = single precision floating-point _fpu = with FPU instructions (default, control program option **--fp-model=-soft**)

To build an STLport library

1. Change to the directory *installdir*\lib\src.stl.

2. Run the program install-tc-libstl-sources.exe. This will extract the library sources.

3. Change to the directory
    [p][tc13][tc13_mmu][tc131][tc16][tc16_mmu][tc16x][tc162]libstl[s]x, depending
    on the library set used by your project.

4. Run the makefile by executing *installdir*\bin\mktc.exe without arguments.

5. Copy the generated C++ library libstl[s]x.a to the corresponding directory
    installdir\lib\[p][tc13][tc13_mmu][tc131][tc16][tc16_mmu][tc16x][tc162].

where,

[tc13]	libraries for TriCore 1.3 architectures
[tc13_mmu]	libraries for TriCore 1.3 architectures that use an MMU
[tc131]	libraries for TriCore 1.3.1 architectures
[tc16]	libraries for TriCore 1.6 architectures
[tc16_mmu]	libraries for TriCore 1.6 architectures that use an MMU
[tc16x]	libraries for TriCore 1.6.x architectures
[tc162]	libraries for TriCore 1.6.2 architectures
[p]	protected libraries for CPU functional problems
[s]	STLport library with exception handling and single precision floating-point

13.1. Library Functions

The tables in the sections below list all library functions, grouped per header file in which they are declared. Some functions are not completely implemented because their implementation depends on the context where your application will run. These functions are for example all I/O related functions. Where possible, these functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment which enables you to debug your application.

A number of wide-character functions are available as C source code, but have not been compiled with the C library. To use complete wide-character functionality, you must recompile the libraries with the macro WCHAR_SUPPORT_ENABLED and keep this macro also defined when compiling your own sources. See C compiler option **--define (-D)**. The easiest way is to adapt the makefile for the library and change the CC line to:

CC = $(PRODDIR)\bin\ctc -DWCHAR_SUPPORT_ENABLED

13.1.1. assert.h

assert(*expr*)   Prints a diagnostic message if NDEBUG is not defined. (Implemented as macro)

13.1.2. complex.h

The complex number *z* is also written as *x+yi* where *x* (the real part) and *y* (the imaginary part) are real numbers of types `float`, `double` or `long double`. The real and imaginary part can be stored in structs or in arrays. This implementation uses arrays because structs may have different alignments.

The header file `complex.h` also defines the following macros for backward compatibility:

```
complex    _Complex    /* C99 keyword */
imaginary  _Imaginary  /* C99 keyword */
```

Parallel sets of functions are defined for double, float and long double. They are respectively named *function*, *function*f, *function*l. All long type functions, though declared in `complex.h`, are implemented as the `double` type variant which nearly always meets the requirement in embedded applications.

This implementation uses the obvious implementation for complex multiplication; and a more sophisticated implementation for division and absolute value calculations which handles underflow, overflow and infinities with more care. The ISO C99 `#pragma CX_LIMITED_RANGE` therefore has no effect.

Trigonometric functions

csin	csinf	csinl	Returns the complex sine of *z*.
ccos	ccosf	ccosl	Returns the complex cosine of *z*.
ctan	ctanf	ctanl	Returns the complex tangent of *z*.
casin	casinf	casinl	Returns the complex arc sine $\sin^{-1}(z)$.
cacos	cacosf	cacosl	Returns the complex arc cosine $\cos^{-1}(z)$.
catan	catanf	catanl	Returns the complex arc tangent $\tan^{-1}(z)$.
csinh	csinhf	csinhl	Returns the complex hyperbolic sine of *z*.
ccosh	ccoshf	ccoshl	Returns the complex hyperbolic cosine of *z*.
ctanh	ctanhf	ctanhl	Returns the complex hyperbolic tangent of *z*.
casinh	casinhf	casinhl	Returns the complex arc hyperbolic sinus of *z*.
cacosh	cacoshf	cacoshl	Returns the complex arc hyperbolic cosine of *z*.
catanh	catanhf	catanhl	Returns the complex arc hyperbolic tangent of *z*.

Exponential and logarithmic functions

cexp	cexpf	cexpl	Returns the result of the complex exponential function e^z.
clog	clogf	clogl	Returns the complex natural logarithm.

Power and absolute-value functions

cabs	cabsf	cabsl	Returns the complex absolute value of *z* (also known as *norm*, *modulus* or *magnitude*).
cpow	cpowf	cpowl	Returns the complex value of *x* raised to the power *y* (x^y) where both *x* and *y* are complex numbers.

`csqrt`	`csqrtf`	`csqrtl`	Returns the complex square root of *z*.

Manipulation functions

`carg`	`cargf`	`cargl`	Returns the argument of z (also known as *phase angle)*.
`cimag`	`cimagf`	`cimagl`	Returns the imaginary part of *z* as a real (respectively as a `double`, `float`, `long double`)
`conj`	`conjf`	`conjl`	Returns the complex conjugate value (the sign of its imaginary part is reversed).
`cproj`	`cprojf`	`cprojl`	Returns the value of the projection of *z* onto the Riemann sphere.
`creal`	`crealf`	`creall`	Returns the real part of *z* as a real (respectively as a `double`, `float`, `long double`)

13.1.3. cstart.h

The header file `cstart.h` controls the system startup code's general settings and register initializations. It contains defines only, no functions.

13.1.4. ctype.h and wctype.h

The header file `ctype.h` declares the following functions which take a character *c* as an integer type argument. The header file `wctype.h` declares parallel wide-character functions which take a character *c* of the `wchar_t` type as argument.

ctype.h	wctype.h	Description
`isalnum`	`iswalnum`	Returns a non-zero value when c is an alphabetic character or a number ([A-Z][a-z][0-9]).
`isalpha`	`iswalpha`	Returns a non-zero value when c is an alphabetic character ([A-Z][a-z]).
`isblank`	`iswblank`	Returns a non-zero value when c is a blank character (tab, space...)
`iscntrl`	`iswcntrl`	Returns a non-zero value when c is a control character.
`isdigit`	`iswditit`	Returns a non-zero value when c is a numeric character ([0-9]).
`isgraph`	`iswgraph`	Returns a non-zero value when c is printable, but not a space.
`islower`	`iswlower`	Returns a non-zero value when c is a lowercase character ([a-z]).
`isprint`	`iswprint`	Returns a non-zero value when c is printable, including spaces.
`ispunct`	`iswpunct`	Returns a non-zero value when c is a punctuation character (such as '.', ',', '!').
`isspace`	`iswspace`	Returns a non-zero value when c is a space type character (space, tab, vertical tab, formfeed, linefeed, carriage return).
`isupper`	`iswupper`	Returns a non-zero value when c is an uppercase character ([A-Z]).
`isxdigit`	`iswxdigit`	Returns a non-zero value when c is a hexadecimal digit ([0-9][A-F][a-f]).

ctype.h	wctype.h	Description
tolower	towlower	Returns c converted to a lowercase character if it is an uppercase character, otherwise c is returned.
toupper	towupper	Returns c converted to an uppercase character if it is a lowercase character, otherwise c is returned.
_tolower	–	Converts c to a lowercase character, does not check if c really is an uppercase character. Implemented as macro. This macro function is not defined in ISO C99.
_toupper	–	Converts c to an uppercase character, does not check if c really is a lowercase character. Implemented as macro. This macro function is not defined in ISO C99.
isascii		Returns a non-zero value when c is in the range of 0 and 127. This function is not defined in ISO C99.
toascii		Converts c to an ASCII value (strip highest bit). This function is not defined in ISO C99.

13.1.5. dbg.h

The header file dbg.h contains the debugger call interface for file system simulation. It contains low level functions. This header file is not defined in ISO C99.

_dbg_trap	Low level function to trap debug events
_argcv(const char *buf,size_t size)	Low level function for command line argument passing

13.1.6. errno.h

int errno	External variable that holds implementation defined error codes.

The following error codes are defined as macros in errno.h:

EPERM	1	Operation not permitted
ENOENT	2	No such file or directory
EINTR	3	Interrupted system call
EIO	4	I/O error
EBADF	5	Bad file number
EAGAIN	6	No more processes
ENOMEM	7	Not enough core
EACCES	8	Permission denied
EFAULT	9	Bad address
EEXIST	10	File exists
ENOTDIR	11	Not a directory
EISDIR	12	Is a directory
EINVAL	13	Invalid argument
ENFILE	14	File table overflow
EMFILE	15	Too many open files

ETXTBSY	16	Text file busy
ENOSPC	17	No space left on device
ESPIPE	18	Illegal seek
EROFS	19	Read-only file system
EPIPE	20	Broken pipe
ELOOP	21	Too many levels of symbolic links
ENAMETOOLONG	22	File name too long

Floating-point errors

EDOM	23	Argument too large
ERANGE	24	Result too large

Errors returned by printf/scanf

ERR_FORMAT	25	Illegal format string for printf/scanf
ERR_NOFLOAT	26	Floating-point not supported
ERR_NOLONG	27	Long not supported
ERR_NOPOINT	28	Pointers not supported

Encoding errors set by functions like fgetwc, getwc, mbrtowc, etc ...

EILSEQ	29	Invalid or incomplete multibyte or wide character

Errors returned by RTOS

ECANCELED	30	Operation canceled
ENODEV	31	No such device

13.1.7. except.h

The header file `except.h` contains the TriCore specific software floating-point exception handling interface definition. This header file is not defined in ISO C99. See the `fptrap` example delivered with the product for an example.

`_fp_install_trap_handler(`
`void (*) (`
`_fp_exception_info_t * )`
`exception )`
	Installs a floating-point trap handler.

`_fp_get_exception_mask(` `void )`	Returns the exception mask.
`_fp_set_exception_mask( int` `)`	Sets the exception mask. A value of 0xFF traps all floating-point exceptions.
`_fp_get_exception_status(` `void )`	Returns the exception status.
`_fp_set_exception_status(` `int )name)`	Sets the exception status.

For each supported exception, a macro is defined. The following exceptions are defined:

EFINVOP	0x01	Invalid operation, for instance 0.0/0.0, 0.0*INF or on a NaN
EFDIVZ	0x02	Division by zero, for instance 12.5/0.0 (not 0.0/0.0)
EFOVFL	0x04	Overflow, when the result of an operation is too large
EFUNFL	0x08	Underflow, when the result of an operation is too small
EFINEXCT	0x10	Inexact
EFALL	0x1f	Combination of all of the above enlisted exceptions

INF means infinite which is the largest absolute floating-point number.

NAN means Not A Number, this is a special notation for an undefined floating-point number.

13.1.8. fcntl.h

The header file `fcntl.h` contains the function `open()`, which calls the low level function `_open()`, and definitions of flags used by the low level function `_open()`. This header file is not defined in ISO C99.

open	Opens a file a file for reading or writing. Calls `_open`. (*FSS implementation*)

13.1.9. fenv.h

Contains mechanisms to control the floating-point environment.

fegetenv	Stores the current floating-point environment.
feholdexept	Saves the current floating-point environment and installs an environment that ignores all floating-point exceptions.
fesetenv	Restores a previously saved (`fegetenv` or `feholdexcept`) floating-point environment.
feupdateenv	Saves the currently raised floating-point exceptions, restores a previously saved floating-point environment and finally raises the saved exceptions.
feclearexcept	Clears the current exception status flags corresponding to the flags specified in the argument.
fegetexceptflag	Stores the current setting of the floating-point status flags.
feraiseexcept	Raises the exceptions represented in the argument. As a result, other exceptions may be raised as well.
fesetexceptflag	Sets the current floating-point status flags.
fetestexcept	Returns the bitwise-OR of the exception macros corresponding to the exception flags which are currently set *and* are specified in the argument.

For each supported exception, a macro is defined. The following exceptions are defined:

FE_DIVBYZERO	FE_INEXACT	FE_INVALID
FE_OVERFLOW	FE_UNDERFLOW	FE_ALL_EXCEPT

`fegetround`	Returns the current rounding direction, represented as one of the values of the rounding direction macros.
`fesetround`	Sets the current rounding directions.

For each supported rounding mode, a macro is defined. The following rounding mode macros are defined:

`FE_TONEAREST`   `FE_UPWARD`   `FE_DOWNWARD`   `FE_TOWARDZERO`

13.1.10. float.h

The header file `float.h` defines the characteristics of the real floating-point types `float`, `_Float16`, `double` and `long double`.

> `float.h` used to contain prototypes for the functions `copysign(f)`, `isinf(f)`, `isfinite(f)`, `isnan(f)` and `scalb(f)`. These functions have accordingly to the ISO C99 standard been moved to the header file `math.h`. See also Section 13.1.17, *math.h and tgmath.h*.

The following functions are only available for ISO C90:

`copysignf(float f,float s)`	Copies the sign of the second argument *s* to the value of the first argument *f* and returns the result.
`copysign(double d,double s)`	Copies the sign of the second argument *s* to the value of the first argument *d* and returns the result.
`isinff(float f)`	Test the variable *f* on being an infinite (IEEE-754) value.
`isinf(double d);`	Test the variable *d* on being an infinite (IEEE-754) value.
`isfinitef(float f)`	Test the variable *f* on being a finite (IEEE-754) value.
`isfinite(double d)`	Test the variable *d* on being a finite (IEEE-754) value.
`isnanf(float f)`	Test the variable *f* on being NaN (Not a Number, IEEE-754) .
`isnan(double d)`	Test the variable *d* on being NaN (Not a Number, IEEE-754) .
`scalbf(float f,int p)`	Returns *f* * 2^*p* for integral values without computing 2^N.
`scalb(double d,int p)`	Returns *d* * 2^*p* for integral values without computing 2^N. (See also `scalbn` in Section 13.1.17, *math.h and tgmath.h*)

13.1.11. inttypes.h and stdint.h

The header files `stdint.h` and `inttypes.h` provide additional declarations for integer types and have various characteristics. The `stdint.h` header file contains basic definitions of integer types of certain sizes, and corresponding sets of macros. This header file clearly refers to the corresponding sections in the ISO C99 standard.
The `inttypes.h` header file includes `stdint.h` and adds portable formatting and conversion functions. Below the conversion functions from `inttypes.h` are listed.

`imaxabs(intmax_t j)`	Returns the absolute value of j

`imaxdiv(intmax_t numer, intmax_t denom)`	Computes `numer/denom`and `numer % denom`. The result is stored in the `quot` and `rem` components of the `imaxdiv_t` structure type.
`strtoimax(const char * restrict nptr, char ** restrict endptr, int base)`	Convert string to maximum sized integer. (Compare `strtoll`)
`strtoumax(const char * restrict nptr, char ** restrict endptr, int base)`	Convert string to maximum sized unsigned integer. (Compare `strtoull`)
`wcstoimax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base)`	Convert wide string to maximum sized integer. (Compare `wcstoll`)
`wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base)`	Convert wide string to maximum sized unsigned integer. (Compare `wcstoull`)

13.1.12. io.h

The header file `io.h` contains prototypes for low level I/O functions. This header file is not defined in ISO C99.

`_close(fd)`	Used by the functions `close` and `fclose`. (*FSS implementation)*
`_lseek(fd,offset,whence)`	Used by all file positioning functions: `fgetpos`, `fseek`, `fsetpos`, `ftell`, `rewind`. (*FSS implementation)*
`_open(fd,flags)`	Used by the functions `fopen` and `freopen`. (*FSS implementation)*
`_read(fd,*buff,cnt)`	Reads a sequence of characters from a file. (*FSS implementation)*
`_unlink(*name)`	Used by the function remove. (*FSS implementation)*
`_write(fd,*buffer,cnt)`	Writes a sequence of characters to a file. (*FSS implementation)*

13.1.13. iso646.h

The header file `iso646.h` adds tokens that can be used instead of regular operator tokens.

```
#define and     &&
#define and_eq  &=
#define bitand  &
#define bitor   |
#define compl   ~
#define not     !
#define not_eq  !=
#define or      ||
#define or_eq   |=
#define xor     ^
#define xor_eq  ^=
```

13.1.14. limits.h

Contains the sizes of integral types, defined as macros.

13.1.15. locale.h

To keep C code reasonable portable across different languages and cultures, a number of facilities are provided in the header file `locale.h`.

```
char *setlocale( int category, const char *locale )
```

The function above changes locale-specific features of the run-time library as specified by the category to change and the name of the locale.

The following categories are defined and can be used as input for this function:

```
LC_ALL       0     LC_NUMERIC   3
LC_COLLATE   1     LC_TIME      4
LC_CTYPE     2     LC_MONETARY  5
```

```
struct lconv *localeconv( void )
```

Returns a pointer to type `struct lconv` with values appropriate for the formatting of numeric quantities according to the rules of the current locale. The `struct lconv` in this header file is conforming the ISO standard.

13.1.16. malloc.h

The header file `malloc.h` contains prototypes for memory allocation functions. This include file is not defined in ISO C99, it is included for backwards compatibility with ISO C90. For ISO C99, the memory allocation functions are part of `stdlib.h`. See Section 13.1.25, *stdlib.h and wchar.h*.

`malloc(size)`	Allocates space for an object with size *size*. The allocated space is not initialized. Returns a pointer to the allocated space.
`calloc(nobj,size)`	Allocates space for n objects with size *size*. The allocated space is initialized with zeros. Returns a pointer to the allocated space.
`free(*ptr)`	Deallocates the memory space pointed to by *ptr* which should be a pointer earlier returned by the `malloc` or `calloc` function.
`realloc(*ptr,size)`	Deallocates the old object pointed to by *ptr* and returns a pointer to a new object with size *size*, while preserving its contents. If the new size is smaller than the old size, some contents at the end of the old region will be discarded. If the new size is larger than the old size, all of the old contents are preserved and any bytes in the new object beyond the size of the old object will have indeterminate values.

13.1.17. math.h and tgmath.h

The header file `math.h` contains the prototypes for many mathematical functions. Before ISO C99, all functions were computed using the double type (the float was automatically converted to double, prior to calculation). In this ISO C99 version, parallel sets of functions are defined for `double`, `float` and `long double`. They are respectively named *function*, *function*f, *function*l. All `long` type functions, though declared in `math.h`, are implemented as the `double` type variant which nearly always meets the requirement in embedded applications.

The header file `tgmath.h` contains parallel type generic math macros whose expansion depends on the used type. `tgmath.h` includes `math.h` and the effect of expansion is that the correct `math.h` functions are called. The type generic macro, if available, is listed in the second column of the tables below.

Trigonometric and hyperbolic functions

math.h			tgmath.h	Description
sin	sinf	sinl	sin	Returns the sine of x.
cos	cosf	cosl	cos	Returns the cosine of x.
tan	tanf	tanl	tan	Returns the tangent of x.
asin	asinf	asinl	asin	Returns the arc sine $\sin^{-1}(x)$ of x.
acos	acosf	acosl	acos	Returns the arc cosine $\cos^{-1}(x)$ of x.
atan	atanf	atanl	atan	Returns the arc tangent $\tan^{-1}(x)$ of x.
atan2	atan2f	atan2l	atan2	Returns the result of: $\tan^{-1}(y/x)$.
sinh	sinhf	sinhl	sinh	Returns the hyperbolic sine of x.
cosh	coshf	coshl	cosh	Returns the hyperbolic cosine of x.
tanh	tanhf	tanhl	tanh	Returns the hyperbolic tangent of x.
asinh	asinhf	asinhl	asinh	Returns the arc hyperbolic sine of x.
acosh	acoshf	acoshl	acosh	Returns the non-negative arc hyperbolic cosine of x.
atanh	atanhf	atanhl	atanh	Returns the arc hyperbolic tangent of x.

Exponential and logarithmic functions

All of these functions are new in ISO C99, except for `exp`, `log` and `log10`.

math.h			tgmath.h	Description
exp	expf	expl	exp	Returns the result of the exponential function e^x.
exp2	exp2f	exp2l	exp2	Returns the result of the exponential function 2^x.
expm1	expm1f	expm1l	expm1	Returns the result of the exponential function e^x-1.
log	logf	logl	log	Returns the natural logarithm $\ln(x)$, $x>0$.
log10	log10f	log10l	log10	Returns the base-10 logarithm of x, $x>0$.
log1p	log1pf	log1pl	log1p	Returns the base-e logarithm of $(1+x)$. $x <> -1$.
log2	log2f	log2l	log2	Returns the base-2 logarithm of x. $x>0$.

math.h			tgmath.h	Description
ilogb	ilogbf	ilogbl	ilogb	Returns the signed exponent of x as an integer. `x>0`.
logb	logbf	logbl	logb	Returns the exponent of `x` as a signed integer in value in floating-point notation. `x > 0`.

frexp, ldexp, modf, scalbn, scalbln

math.h			tgmath.h	Description
frexp	frexpf	frexpl	frexp	Splits a float x into fraction f and exponent n, so that: $f = 0.0$ or $0.5 \le \lvert f \rvert \le 1.0$ and $f*2^{n} = x$. Returns f, stores n.
ldexp	ldexpf	ldexpl	ldexp	Inverse of `frexp`. Returns the result of $x*2^{n}$. (x and n are both arguments).
modf	modff	modfl	–	Splits a float x into fraction f and integer n, so that: $\lvert f \rvert < 1.0$ and $f+n=x$. Returns f, stores n.
scalbn	scalbnf	scalbnl	scalbn	Computes the result of $x*\text{FLT\_RADIX}^{n}$. efficiently, not normally by computing FLT_RADIX^{n} explicitly.
scalbln	scalblnf	scalblnl	scalbln	Same as `scalbn` but with argument n as `long int`.

Rounding functions

math.h			tgmath.h	Description
ceil	ceilf	ceill	ceil	Returns the smallest integer not less than `x`, as a double.
floor	floorf	floorl	floor	Returns the largest integer not greater than `x`, as a double.
rint	rintf	rintl	rint	Returns the rounded integer value as an `int` according to the current rounding direction. See `fenv.h`.
lrint	lrintf	lrintl	lrint	Returns the rounded integer value as a `long int` according to the current rounding direction. See `fenv.h`.
llrint	llrintf	llrintl	llrint	Returns the rounded integer value as a `long long int` according to the current rounding direction. See `fenv.h`.
nearbyint	nearbyintf	nearbyintl	nearbyint	Returns the rounded integer value as a floating-point according to the current rounding direction. See `fenv.h`.
round	roundf	roundl	round	Returns the nearest integer value of `x` as int.
lround	lroundf	lroundl	lround	Returns the nearest integer value of `x` as long int.
llround	lroundf	llroundl	llround	Returns the nearest integer value of `x` as long long int.
trunc	truncf	truncl	trunc	Returns the truncated integer value `x`.

Remainder after division

math.h			tgmath.h	Description
fmod	fmodf	fmodl	fmod	Returns the remainder `r` of `x-ny`. `n` is chosen as `trunc(`x/y`)`. `r` has the same sign as `x`.

math.h			tgmath.h	Description
remainder	remainderf	remainderl	remainder	Returns the remainder r of $x-ny$. n is chosen as $trunc(x/y)$. r may not have the same sign as x.
remquo	remquof	remquol	remquo	Same as remainder. In addition, the argument *quo is given a specific value (see ISO).

Power and absolute-value functions

math.h			tgmath.h	Description
cbrt	cbrtf	cbrtl	cbrt	Returns the real cube root of x ($=x^{1/3}$).
fabs	fabsf	fabsl	fabs	Returns the absolute value of x ($|x|$). (abs, labs, llabs, div, ldiv, lldiv are defined in stdlib.h)
fma	fmaf	fmal	fma	Floating-point multiply add. Returns $x*y+z$.
hypot	hypotf	hypotl	hypot	Returns the square root of x^2+y^2.
pow	powf	powl	power	Returns x raised to the power y (x^y). [1]
sqrt	sqrtf	sqrtl	sqrt	Returns the non-negative square root of x. $x \geq 0$.

[1]With compiler option **--fp-model=-fastlib** more precise versions of pow, powf and powl are used when the arguments have no fractional part. The macro __PRECISE_LIB_FP__ is defined. By default, less precise versions are used.

Manipulation functions: copysign, nan, nextafter, nexttoward

math.h			tgmath.h	Description
copysign	copysignf	copysignll	copysign	Returns the value of x with the sign of y.
nan	nanf	nanl	–	Returns a quiet NaN, if available, with content indicated through *tagp*.
nextafter	nextafterf	nextafterl	nextafter	Returns the next representable value in the specified format after x in the direction of y. Returns y is $x=y$.
nexttoward	nexttowardf	nexttowardl	nexttoward	Same as nextafter, except that the second argument in all three variants is of type long double. Returns y if $x=y$.

Positive difference, maximum, minimum

math.h			tgmath.h	Description
fdim	fdimf	fdiml	fdim	Returns the positive difference between: $|x-y|$.
fmax	fmaxf	fmaxl	fmax	Returns the maximum value of their arguments.
fmin	fminf	fminl	fmin	Returns the minimum value of their arguments.

Error and gamma

math.h			tgmath.h	Description
`erf`	`erff`	`erfl`	`erf`	Computes the error function of x.
`erfc`	`erfcf`	`erfcl`	`erc`	Computes the complementary error function of x.
`lgamma`	`lgammaf`	`lgammal`	`lgamma`	Computes the $*\log_e|\Gamma(x)|$
`tgamma`	`tgammaf`	`tgammal`	`tgamma`	Computes $\Gamma(x)$

Comparison macros

The next are implemented as macros. For any ordered pair of numeric values exactly one of the relationships - *less*, *greater*, and *equal* - is true. These macros are type generic and therefore do not have a parallel function in `tgmath.h`. All arguments must be expressions of real-floating type.

math.h	tgmath.h	Description
`isgreater`	–	Returns the value of `(x) > (y)`
`isgreaterequal`	–	Returns the value of `(x) >= (y)`
`isless`	–	Returns the value of `(x) < (y)`
`islessequal`	–	Returns the value of `(x) <= (y)`
`islessgreater`	–	Returns the value of `(x) < (y) || (x) > (y)`
`isunordered`	–	Returns 1 if its arguments are unordered, 0 otherwise.

Classification macros

The next are implemented as macros. These macros are type generic and therefore do not have a parallel function in `tgmath.h`. All arguments must be expressions of real-floating type.

math.h	tgmath.h	Description
`fpclassify`	–	Returns the class of its argument: `FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL` or `FP_ZERO`
`isfinite`	–	Returns a nonzero value if and only if its argument has a finite value
`isinf`	–	Returns a nonzero value if and only if its argument has an infinite value
`isnan`	–	Returns a nonzero value if and only if its argument has NaN value.
`isnormal`	–	Returns a nonzero value if an only if its argument has a normal value.
`signbit`	–	Returns a nonzero value if and only if its argument value is negative.

13.1.18. setjmp.h

The `setjmp` and `longjmp` in this header file implement a primitive form of non-local jumps, which may be used to handle exceptional situations. This facility is traditionally considered more portable than `signal.h`

`int setjmp(jmp_buf env)`	Records its caller's environment in `env` and returns 0.
`void longjmp(jmp_buf env, int status)`	Restores the environment previously saved with a call to `setjmp()`.

13.1.19. signal.h

Signals are possible asynchronous events that may require special processing. Each signal is named by a number. The following signals are defined:

`SIGINT`	1	Receipt of an interactive attention signal
`SIGILL`	2	Detection of an invalid function message
`SIGFPE`	3	An erroneous arithmetic operation (for example, zero divide, `overflow`)
`SIGSEGV`	4	An invalid access to storage
`SIGTERM`	5	A termination request sent to the program
`SIGABRT`	6	Abnormal termination, such as is initiated by the `abort` function

The next function sends the signal *sig* to the program:

```
int raise(int sig)
```

The next function determines how subsequent signals will be handled:

```
signalfunction *signal (int, signalfunction *);
```

The first argument specifies the signal, the second argument points to the signal-handler function or has one of the following values:

`SIG_DFL`	Default behavior is used
`SIG_IGN`	The signal is ignored

The function returns the previous value of `signalfunction` for the specific signal, or `SIG_ERR` if an error occurs.

13.1.20. stdarg.h

The facilities in this header file gives you a portable way to access variable arguments lists, such as needed for as `fprintf` and `vfprintf`. `va_copy` is new in ISO C99. This header file contains the following macros:

`va_arg(va_list ap,type)`	Returns the value of the next argument in the variable argument list. Its return type has the type of the given argument `type`. A next call to this macro will return the value of the next argument.
`va_copy(va_list dest, va_list src)`	This macro duplicates the current state of `src` in `dest`, creating a second pointer into the argument list. After this call, va_arg() may be used on `src` and `dest` independently.
`va_end(va_list ap)`	This macro must be called after the arguments have been processed. It should be called before the function using the macro 'va_start' is terminated.
`va_start(va_list ap, lastarg)`	This macro initializes `ap`. After this call, each call to va_arg() will return the value of the next argument. In our implementation, `va_list` cannot contain any bit type variables. Also the given argument `lastarg` must be the last non-bit type argument in the list.

13.1.21. stdbool.h

This header file contains the following macro definitions. These names for boolean type and values are consistent with C++. You are allowed to `#undefine` or redefine the macros below.

```
#define bool                         _Bool
#define true                         1
#define false                        0
#define __bool_true_false_are_defined  1
```

13.1.22. stddef.h

This header file defines the types for common use:

`ptrdiff_t`	Signed integer type of the result of subtracting two pointers.
`size_t`	Unsigned integral type of the result of the `sizeof` operator.
`wchar_t`	Integer type to represent character codes in large character sets.

Besides these types, the following macros are defined:

`NULL`	Expands to the null pointer constant `(void *) 0` for C or 0 (zero) for C++.
`offsetof(_type, _member)`	Expands to an integer constant expression with type `size_t` that is the offset in bytes of `_member` within structure type `_type`.

13.1.23. stdint.h

See Section 13.1.11, *inttypes.h and stdint.h*

13.1.24. stdio.h and wchar.h

Types

The header file `stdio.h` contains functions for performing input and output. A number of functions also have a parallel wide character function or macro, defined in `wchar.h`. The header file `wchar.h` also includes `stdio.h`.

In the C language, many I/O facilities are based on the concept of streams. The `stdio.h` header file defines the data type **FILE** which holds the information about a stream. A FILE object is created with the function `fopen`. The pointer to this object is used as an argument in many of the in this header file. The FILE object can contain the following information:

- the current position within the stream

- pointers to any associated buffers

- indications of for read/write errors

- end of file indication

The header file also defines type `fpos_t` as an `unsigned long`.

Macros

stdio.h	Description
NULL	Expands to the null pointer constant `(void *) 0` for C or 0 (zero) for C++.
BUFSIZ	Size of the buffer used by the `setbuf`/`setvbuf` function: 512
EOF	End of file indicator. Expands to -1.
WEOF	End of file indicator. Expands to UINT_MAX (defined in `limits.h`) NOTE: WEOF need not to be a negative number as long as its value does not correspond to a member of the wide character set. (Defined in `wchar.h`).
FOPEN_MAX	Number of files that can be opened simultaneously: 10
FILENAME_MAX	Maximum length of a filename: 100
_IOFBF _IOLBF _IONBF	Expand to an integer expression, suitable for use as argument to the `setvbuf` function.
L_tmpnam	Size of the string used to hold temporary file names: 8 (tmp*xxxxx*)
TMP_MAX	Maximum number of unique temporary filenames that can be generated: 0x8000
SEEK_CUR SEEK_END SEEK_SET	Expand to an integer expression, suitable for use as the third argument to the `fseek` function.
stderr stdin stdout	Expressions of type "pointer to FILE" that point to the FILE objects associated with standard error, input and output streams.

File access

stdio.h	Description
fopen(*name*,*mode*)	Opens a file for a given mode. Available modes are:

`"r"`	read; open text file for reading	
`"w"`	write; create text file for writing; if the file already exists, its contents is discarded	
`"a"`	append; open existing text file or create new text file for writing at end of file	
`"r+"`	open text file for update; reading and writing	
`"w+"`	create text file for update; previous contents if any is discarded	
`"a+"`	append; open or create text file for update, writes at end of file	

(*FSS implementation*)

stdio.h	Description
fclose(*name*)	Flushes the data stream and closes the specified file that was previously opened with fopen. (*FSS implementation*)
fflush(*name*)	If stream is an output stream, any buffered but unwritten date is written. Else, the effect is undefined. (*FSS implementation*)
freopen(*name*,*mode*, *stream*)	Similar to fopen, but rather than generating a new value of type FILE *, the existing value is associated with a new stream. (*FSS implementation*)
setbuf(*stream*,*buffer*)	If buffer is NULL, buffering is turned off for the stream. Otherwise, setbuf is equivalent to: (void) setvbuf(*stream*,*buffer*,_IOFBF,BUFSIZ).
setvbuf(*stream*,*buffer*,*mode*, *size*)	Controls buffering for the *stream*; this function must be called before reading or writing. *Mode* can have the following values: `_IOFBF` causes full buffering `_IOLBF` causes line buffering of text files `_IONBF` causes no buffering. If *buffer* is not NULL, it will be used as a buffer; otherwise a buffer will be allocated. *size* determines the buffer size.

Formatted input/output

The format string of **printf** related functions can contain plain text mixed with conversion specifiers. Each conversion specifier should be preceded by a '%' character. The conversion specifier should be built in order:

- Flags (in any order):

  −   specifies left adjustment of the converted argument.

  +   a number is always preceded with a sign character.
          + has higher precedence than space.

  space a negative number is preceded with a sign, positive numbers with a space.

  0   specifies padding to the field width with zeros (only for numbers).

specifies an alternate output form. For o, the first digit will be zero. For x or X, "0x" and "0X" will be prefixed to the number. For e, E, f, g, G, the output always contains a decimal point, trailing zeros are not removed.

- A number specifying a minimum field width. The converted argument is printed in a field with at least the length specified here. If the converted argument has fewer characters than specified, it will be padded at the left side (or at the right when the flag '-' was specified) with spaces. Padding to numeric fields will be done with zeros when the flag '0' is also specified (only when padding left). Instead of a numeric value, also '*' may be specified, the value is then taken from the next argument, which is assumed to be of type `int`.

- A period. This separates the minimum field width from the precision.

- A number specifying the maximum length of a string to be printed. Or the number of digits printed after the decimal point (only for floating-point conversions). Or the minimum number of digits to be printed for an integer conversion. Instead of a numeric value, also '*' may be specified, the value is then taken from the next argument, which is assumed to be of type `int`.

- A length modifier 'h', 'hh', 'l', 'll', 'L', 'j', 'z' or 't'. 'h' indicates that the argument is to be treated as a `short` or `unsigned short`. 'hh' indicates that the argument is to be treated as a `char` or `unsigned char`. 'l' should be used if the argument is a `long` integer, 'll' for a `long long`. 'L' indicates that the argument is a `long double`. 'j' indicates a pointer to `intmax_t` or `uintmax_t`, 'z' indicates a pointer to `size_t` and 't' indicates a pointer to `ptrdiff_t`.

Flags, length specifier, period, precision and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined:

Character	Printed as
d, i	int, signed decimal
o	int, unsigned octal
x, X	int, unsigned hexadecimal in lowercase or uppercase respectively
u	int, unsigned decimal
c	int, single character (converted to unsigned char)
s	char *, the characters from the string are printed until a NULL character is found. When the given precision is met before, printing will also stop
f, F	double
e, E	double
g, G	double
a, A	double
n	int *, the number of characters written so far is written into the argument. This should be a pointer to an integer in default memory. No value is printed.
p	pointer
r, lr	__fract, __lfract

Character	Printed as
R, IR	__accum, __laccum
%	No argument is converted, a '%' is printed.

printf conversion characters

All arguments to the **scanf** related functions should be pointers to variables (in default memory) of the type which is specified in the format string.

The format string can contain :

- Blanks or tabs, which are skipped.

- Normal characters (not '%'), which should be matched exactly in the input stream.

- Conversion specifications, starting with a '%' character.

Conversion specifications should be built as follows (in order) :

- A '*', meaning that no assignment is done for this field.

- A number specifying the maximum field width.

- The conversion characters `d`, `i`, `n`, `o`, `u` and `x` may be preceded by 'h' if the argument is a pointer to `short` rather than `int`, or by 'hh' if the argument is a pointer to `char`, or by 'l' (letter ell) if the argument is a pointer to `long` or by 'll' for a pointer to `long long`, 'j' for a pointer to `intmax_t` or `uintmax_t`, 'z' for a pointer to `size_t` or 't' for a pointer to `ptrdiff_t`. The conversion characters `e`, `f`, and `g` may be preceded by 'l' if the argument is a pointer to `double` rather than `float`, and by 'L' for a pointer to a `long double`.

- A conversion specifier. '*', maximum field width and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Length specifier and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Character	Scanned as
d	int, signed decimal.
i	int, the integer may be given octal (i.e. a leading 0 is entered) or hexadecimal (leading "0x" or "0X"), or just decimal.
o	int, unsigned octal.
u	int, unsigned decimal.
x	int, unsigned hexadecimal in lowercase or uppercase.
c	single character (converted to unsigned char).
s	char *, a string of non white space characters. The argument should point to an array of characters, large enough to hold the string and a terminating NULL character.
f, F	float

Character	Scanned as
e, E	float
g, G	float
a, A	float
n	int *, the number of characters written so far is written into the argument. No scanning is done.
p	pointer; hexadecimal value which must be entered without 0x- prefix.
r, lr	__fract, __lfract
R, lR	__accum, __laccum
[...]	Matches a string of input characters from the set between the brackets. A NULL character is added to terminate the string. Specifying [ ]...] includes the ']' character in the set of scanning characters.
[^...]	Matches a string of input characters not in the set between the brackets. A NULL character is added to terminate the string. Specifying [^]...] includes the ']' character in the set.
%	Literal '%', no assignment is done.

scanf conversion characters

stdio.h	wchar.h	Description
`fscanf(`*stream*, *format*`, ...)`	`fwscanf(`*stream*, *format*`, ...)`	Performs a formatted read from the given *stream*. Returns the number of items converted successfully. (*FSS implementation*)
`scanf(`*format*`,...)`	`wscanf(`*format*`, ...)`	Performs a formatted read from `stdin`. Returns the number of items converted successfully. (*FSS implementation*)
`sscanf(*`*s*, *format*, `...)`	`swscanf(*`*s*, *format*, `...)`	Performs a formatted read from the string *s*. Returns the number of items converted successfully.
`vfscanf(`*stream*, *format*, *arg*`)`	`vfwscanf(`*stream*, *format*, *arg*`)`	Same as `fscanf`/`fwscanf`, but extra arguments are given as variable argument list *arg*. (See Section 13.1.20, *stdarg.h*)
`vscanf(`*format*, *arg*`)`	`vwscanf(`*format*, *arg*`)`	Same as `sscanf`/`swscanf`, but extra arguments are given as variable argument list *arg*. (See Section 13.1.20, *stdarg.h*)
`vsscanf(*`*s*, *format*, *arg*`)`	`vswscanf(*`*s*, *format*, *arg*`)`	Same as `scanf`/`wscanf`, but extra arguments are given as variable argument list *arg*. (See Section 13.1.20, *stdarg.h*)
`fprintf(`*stream*, *format*`, ...)`	`fwprintf(`*stream*, *format*`, ...)`	Performs a formatted write to the given *stream*. Returns EOF/WEOF on error. (*FSS implementation*)
`printf(`*format*`, ...)`	`wprintf(`*format*`, ...)`	Performs a formatted write to the stream `stdout`. Returns EOF/WEOF on error. (*FSS implementation*)

stdio.h	wchar.h	Description
sprintf(*s, *format*, ...)	–	Performs a formatted write to string *s*. Returns EOF/WEOF on error.
snprintf(*s, *n*, *format*, ...)	swprintf(*s, *n*, *format*, ...)	Same as sprintf, but n specifies the maximum number of characters (including the terminating null character) to be written.
vfprintf(*stream*, *format*, *arg*)	vfwprintf(*stream*, *format*, *arg*)	Same as fprintf/fwprintf, but extra arguments are given as variable argument list *arg*. (See Section 13.1.20, *stdarg.h*) (*FSS implementation)*
vprintf(*format*, *arg*)	vwprintf(*format*, *arg*)	Same as printf/wprintf, but extra arguments are given as variable argument list *arg*. (See Section 13.1.20, *stdarg.h*) (*FSS implementation)*
vsprintf(*s, *format*, *arg*)	vswprintf(*s, *format*, *arg*)	Same as sprintf/swprintf, but extra arguments are given as variable argument list *arg*. (See Section 13.1.20, *stdarg.h*)

The C library functions printf(), fprintf(), vfprintf(), vsprintf(), ... call one single function, _doprint(), that deals with the format string and arguments. The same applies to all scanf type functions, which call the function _doscan(), and also for the wprintf and wscanf type functions which call _dowprint() and _dowscan() respectively. The C library contains three versions of these routines: int, long and long long versions. If you use floating-point the formatter function for floating-point _doflt() or _dowflt() is called. Depending on the formatting arguments you use, the correct routine is used from the library. Of course the larger the version of the routine the larger your produced code will be.

Note that when you call any of the printf/scanf routines indirectly, the arguments are not known and always the long long version with floating-point support is used from the library.

Example:

```
#include <stdio.h>

long L;

void main(void)
{
    printf( "This is a long: %ld\n", L );
}
```

The linker extracts the long version without floating-point support from the library.

See also the description of #pragma weak in Section 1.8, *Pragmas to Control the Compiler*.

Character input/output

stdio.h	wchar.h	Description
fgetc(*stream*)	fgetwc(*stream*)	Reads one character from *stream*. Returns the read character, or EOF/WEOF on error. (*FSS implementation*)
getc(*stream*)	getwc(*stream*)	Same as fgetc/fgetwc except that is implemented as a macro. (*FSS implementation*) NOTE: Currently #defined as getchar()/getwchar() because FILE I/O is not supported. Returns the read character, or EOF/WEOF on error.
getchar(stdin)	getwchar(stdin)	Reads one character from the stdin stream. Returns the character read or EOF/WEOF on error. Implemented as macro. (*FSS implementation*)
fgets(**s*, *n*, *stream*)	fgetws(**s*, *n*, *stream*)	Reads at most the next *n*-1 characters from the *stream* into array *s* until a newline is found. Returns s or NULL or EOF/WEOF on error. (*FSS implementation*)
gets(**s*, *n*, stdin)	–	Reads at most the next *n*-1 characters from the stdin stream into array *s*. A newline is ignored. Returns *s* or NULL or EOF/WEOF on error. (*FSS implementation*)
ungetc(*c*, *stream*)	ungetwc(*c*, *stream*)	Pushes character *c* back onto the input *stream*. Returns EOF/WEOF on error.
fputc(*c*, *stream*)	fputwc(*c*, *stream*)	Put character *c* onto the given *stream*. Returns EOF/WEOF on error. (*FSS implementation*)
putc(*c*, *stream*)	putwc(*c*, *stream*)	Same as fpuc/fputwc except that is implemented as a macro. (*FSS implementation*)
putchar(*c*, stdout)	putwchar(*c*, stdout)	Put character *c* onto the stdout stream. Returns EOF/WEOF on error. Implemented as macro. (*FSS implementation*)
fputs(**s*, *stream*)	fputws(**s*, *stream*)	Writes string *s* to the given *stream*. Returns EOF/WEOF on error. (*FSS implementation*)
puts(**s*)	–	Writes string *s* to the stdout stream. Returns EOF/WEOF on error. (*FSS implementation*)

Direct input/output

stdio.h	Description
fread(ptr,size,nobj,stream)	Reads *nobj* members of *size* bytes from the given *stream* into the array pointed to by *ptr*. Returns the number of elements successfully read. (*FSS implementation*)

stdio.h	Description
`fwrite(ptr,size,nobj,stream)`	Writes *nobj* members of *size* bytes from to the array pointed to by *ptr* to the given *stream*. Returns the number of elements successfully written. (*FSS implementation*)

Random access

stdio.h	Description
`fseek(stream, offset, origin)`	Sets the position indicator for *stream*. (*FSS implementation*)

When repositioning a binary file, the new position *origin* is given by the following macros:

`SEEK_SET` 0 *offset* characters from the beginning of the file
`SEEK_CUR` 1 *offset* characters from the current position in the file
`SEEK_END` 2 *offset* characters from the end of the file

`ftell(stream)`	Returns the current file position for *stream*, or -1L on error. (*FSS implementation*)
`rewind(stream)`	Sets the file position indicator for the *stream* to the beginning of the file. This function is equivalent to: `(void) fseek(stream,0L,SEEK_SET);` `clearerr(stream);` (*FSS implementation*)
`fgetpos(stream,pos)`	Stores the current value of the file position indicator for *stream* in the object pointed to by *pos*. (*FSS implementation*)
`fsetpos(stream,pos)`	Positions *stream* at the position recorded by `fgetpos` in **pos*. (*FSS implementation*)

Operations on files

stdio.h	Description
`remove(file)`	Removes the named file, so that a subsequent attempt to open it fails. Returns a non-zero value if not successful.
`rename(old,new)`	Changes the name of the file from old name to new name. Returns a non-zero value if not successful.
`tmpfile()`	Creates a temporary file of the mode "`wb+`" that will be automatically removed when closed or when the program terminates normally. Returns a `file` pointer.
`tmpnam(buffer)`	Creates new file names that do not conflict with other file names currently in use. The new file name is stored in a *buffer* which must have room for L_tmpnam characters. Returns a pointer to the temporary name. The file names are created in the current directory and all start with "tmp". At most TMP_MAX unique file names can be generated.

Error handling

stdio.h	Description
clearerr(*stream*)	Clears the end of file and error indicators for stream.
ferror(*stream*)	Returns a non-zero value if the error indicator for stream is set.
feof(*stream*)	Returns a non-zero value if the end of file indicator for stream is set.
perror(**s*)	Prints *s* and the error message belonging to the integer errno. (See Section 13.1.6, *errno.h*)

13.1.25. stdlib.h and wchar.h

The header file stdlib.h contains general utility functions which fall into the following categories (Some have parallel wide-character, declared in wchar.h)

- Numeric conversions

- Random number generation

- Memory management

- Environment communication

- Searching and sorting

- Integer arithmetic

- Multibyte/wide character and string conversions.

Macros

EXIT_SUCCES 0 EXIT_FAILURE 1	Predefined exit codes that can be used in the exit function.
RAND_MAX 32767	Highest number that can be returned by the rand/srand function.
MB_CUR_MAX 1	Maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category LC_CTYPE, see Section 13.1.15, *locale.h*).

Numeric conversions

The following functions convert the initial portion of a string **s* to a double, int, long int and long long int value respectively.

```
double      atof(*s)
int         atoi(*s)
long        atol(*s)
long long   atoll(*s)
```

The following functions convert the initial portion of the string **s* to a float, double and long double value respectively. **endp* will point to the first character not used by the conversion.

stdlib.h	wchar.h
`float        strtof(*s,**endp)`	`float         wcstof(*s,**endp)`
`double       strtod(*s,**endp)`	`double        wcstod(*s,**endp)`
`long double strtold(*s,**endp)`	`long double wcstold(*s,**endp)`

The following functions convert the initial portion of the string **s* to a `long`, `long long`, `unsigned long` and `unsigned long long` respectively. Base specifies the radix. **endp* will point to the first character not used by the conversion.

stdlib.h	wchar.h
`long strtol (*s,**endp,base)`	`long wcstol (*s,**endp,base)`
`long long strtoll`	`long long wcstoll`
`         (*s,**endp,base)`	`               (*s,**endp,base)`
`unsigned long strtoul`	`unsigned long wcstoul`
`         (*s,**endp,base)`	`               (*s,**endp,base)`
`unsigned long long strtoull`	`unsigned long long wcstoull`
`         (*s,**endp,base)`	`               (*s,**endp,base)`

Random number generation

`rand`   Returns a pseudo random integer in the range 0 to RAND_MAX.

`srand(seed)` Same as rand but uses *seed* for a new sequence of pseudo random numbers.

Memory management

`malloc(size)`   Allocates space for an object with size *size*.
The allocated space is not initialized. Returns a pointer to the allocated space.

`calloc(nobj,size)`   Allocates space for n objects with size *size*.
The allocated space is initialized with zeros. Returns a pointer to the allocated space.

`free(*ptr)`   Deallocates the memory space pointed to by *ptr* which should be a pointer earlier returned by the `malloc` or `calloc` function.

`realloc(*ptr,size)` Deallocates the old object pointed to by *ptr* and returns a pointer to a new object with size *size*, while preserving its contents.
If the new size is smaller than the old size, some contents at the end of the old region will be discarded. If the new size is larger than the old size, all of the old contents are preserved and any bytes in the new object beyond the size of the old object will have indeterminate values.

Environment communication

`abort()`	Causes abnormal program termination. If the signal SIGABRT is caught, the signal handler may take over control. (See Section 13.1.19, *signal.h*).
`atexit(*func)`	*func* points to a function that is called (without arguments) when the program normally terminates.
`exit(status)`	Causes normal program termination. Acts as if `main()` returns with status as the return value. Status can also be specified with the predefined macros EXIT_SUCCES or EXIT_FAILURE.
`_Exit(status)`	Same as `exit`, but not registered by the `atexit` function or signal handlers registered by the `signal` function are called.
`getenv(*s)`	Searches an environment list for a string *s*. Returns a pointer to the contents of *s*. NOTE: this function is not implemented because there is no OS.
`system(*s)`	Passes the string *s* to the environment for execution. NOTE: this function is not implemented because there is no OS.

Searching and sorting

`bsearch(*key, *base, n, size, *cmp)`	This function searches in an array of *n* members, for the object pointed to by *key*. The initial base of the array is given by *base*. The size of each member is specified by *size*. The given array must be sorted in ascending order, according to the results of the function pointed to by *cmp*. Returns a pointer to the matching member in the array, or NULL when not found.
`qsort(*base, n, size, *cmp)`	This function sorts an array of *n* members using the quick sort algorithm. The initial base of the array is given by *base*. The size of each member is specified by *size*. The array is sorted in ascending order, according to the results of the function pointed to by *cmp*.

Integer arithmetic

`int abs(j)`
`long labs(j)`
`long long llabs(j)`
	Compute the absolute value of an `int`, `long int`, and `long long int` *j* respectively.

`div_t div(x,y)`
`ldiv_t ldiv(x,y)`
`lldiv_t lldiv(x,y)`
	Compute *x/y* and *x%y* in a single operation. *X* and *y* have respectively type `int`, `long int` and `long long int`. The result is stored in the members `quot` and `rem` of struct `div_t`, `ldiv_t` and `lldiv_t` which have the same types.

Multibyte/wide character and string conversions

`mblen(*s,n)`	Determines the number of bytes in the multi-byte character pointed to by *s*. At most *n* characters will be examined. (See also `mbrlen` in Section 13.1.29, *wchar.h*).
`mbtowc(*pwc,*s,n)`	Converts the multi-byte character in *s* to a wide-character code and stores it in pwc. At most *n* characters will be examined.

`wctomb(*s,wc)`   Converts the wide-character *wc* into a multi-byte representation and stores it in the string pointed to by *s*. At most MB_CUR_MAX characters are stored.

`mbstowcs(*pwcs,*s,n)` Converts a sequence of multi-byte characters in the string pointed to by *s* into a sequence of wide characters and stores at most *n* wide characters into the array pointed to by *pwcs*. (See also `mbsrtowcs` in Section 13.1.29, *wchar.h*).

`wcstombs(*s,*pwcs,n)` Converts a sequence of wide characters in the array pointed to by *pwcs* into multi-byte characters and stores at most *n* multi-byte characters into the string pointed to by *s*. (See also `wcsrtowmb` in Section 13.1.29, *wchar.h*).

13.1.26. string.h and wchar.h

This header file provides numerous functions for manipulating strings. By convention, strings in C are arrays of characters with a terminating null character. Most functions therefore take arguments of type `*char`. However, many functions have also parallel wide-character functions which take arguments of type `*wchar_t`. These functions are declared in `wchar.h`.

Copying and concatenation functions

string.h	wchar.h	Description
`memcpy(*s1,*s2,n)`	`wmemcpy(*s1,*s2,n)`	Copies *n* characters from *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.
`memmove(*s1,*s2,n)`	`wmemmove(*s1,*s2,n)`	Same as `memcpy`, but overlapping strings are handled correctly. Returns *s1.
`strcpy(*s1,*s2)`	`wcscpy(*s1,*s2)`	Copies *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.
`strncpy(*s1,*s2,n)`	`wcsncpy(*s1,*s2,n)`	Copies not more than *n* characters from *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.
`strcat(*s1,*s2)`	`wcscat(*s1,*s2)`	Appends a copy of *s2 to *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.
`strncat(*s1,*s2,n)`	`wcsncat(*s1,*s2,n)`	Appends not more than *n* characters from *s2 to *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.

Comparison functions

string.h	wchar.h	Description
`memcmp(*s1,*s2,n)`	`wmemcmp(*s1,*s2,n)`	Compares the first *n* characters of *s1 to the first *n* characters of *s2. Returns < 0 if *s1 < *s2, 0 if *s1 = = *s2, or > 0 if *s1 > *s2.
`strcmp(*s1,*s2)`	`wcscmp(*s1,*s2)`	Compares string *s1 to *s2. Returns < 0 if *s1 < *s2, 0 if *s1 = = *s2, or > 0 if *s1 > *s2.
`strncmp(*s1,*s2,n)`	`wcsncmp(*s1,*s2,n)`	Compares the first *n* characters of *s1 to the first *n* characters of *s2. Returns < 0 if *s1 < *s2, 0 if *s1 = = *s2, or > 0 if *s1 > *s2.

string.h	wchar.h	Description
strcoll(*s1,*s2)	wcscoll(*s1,*s2)	Performs a local-specific comparison between string *s1 and string *s2 according to the LC_COLLATE category of the current locale. Returns < 0 if *s1 < *s2, 0 if *s1 = = *s2, or > 0 if *s1 > *s2. (See Section 13.1.15, *locale.h*)
strxfrm(*s1,*s2,n)	wcsxfrm(*s1,*s2,n)	Transforms (a local) string *s2 so that a comparison between transformed strings with strcmp gives the same result as a comparison between non-transformed strings with strcoll. Returns the transformed string *s1.

Search functions

string.h	wchar.h	Description
memchr(*s,c,n)	wmemchr(*s,c,n)	Checks the first *n* characters of *s on the occurrence of character *c*. Returns a pointer to the found character.
strchr(*s,c)	wcschr(*s,c)	Returns a pointer to the first occurrence of character *c* in *s or the null pointer if not found.
strrchr(*s,c)	wcsrchr(*s,c)	Returns a pointer to the last occurrence of character *c* in *s or the null pointer if not found.
strspn(*s,*set)	wcsspn(*s,*set)	Searches *s for a sequence of characters specified in *set*. Returns the length of the first sequence found.
strcspn(*s,*set)	wcscspn(*s,*set)	Searches *s for a sequence of characters *not* specified in *set*. Returns the length of the first sequence found.
strpbrk(*s,*set)	wcspbrk(*s,*set)	Same as strspn/wcsspn but returns a pointer to the first character in *s that also is specified in *set*.
strstr(*s,*sub)	wcsstr(*s,*sub)	Searches for a substring *sub* in *s. Returns a pointer to the first occurrence of *sub* in *s.
strtok(*s,*dlm)	wcstok(*s,*dlm)	A sequence of calls to this function breaks the string *s into a sequence of tokens delimited by a character specified in *dlm*. The token found in *s is terminated with a null character. Returns a pointer to the first position in *s of the token.

Miscellaneous functions

string.h	wchar.h	Description
memset(*s,c,n)	wmemset(*s,c,n)	Fills the first *n* bytes of *s with character *c* and returns *s.
strerror(errno)	–	Typically, the values for errno come from int errno. This function returns a pointer to the associated error message. (See also Section 13.1.6, *errno.h*)
strlen(*s)	wcslen(*s)	Returns the length of string *s.

13.1.27. time.h and wchar.h

The header file `time.h` provides facilities to retrieve and use the (calendar) date and time, and the process time. Time can be represented as an integer value, or can be broken-down in components. Two arithmetic data types are defined which are capable of holding the integer representation of times:

```
clock_t unsigned long long
time_t unsigned long
```

The type `struct tm` below is defined according to ISO C99 with one exception: this implementation does not support leap seconds. The `struct tm` type is defines as follows:

```
struct tm
{
  int    tm_sec;        /* seconds after the minute - [0, 59]    */
  int    tm_min;        /* minutes after the hour - [0, 59]      */
  int    tm_hour;       /* hours since midnight - [0, 23]        */
  int    tm_mday;       /* day of the month - [1, 31]            */
  int    tm_mon;        /* months since January - [0, 11]        */
  int    tm_year;       /* year since 1900                       */
  int    tm_wday;       /* days since Sunday - [0, 6]            */
  int    tm_yday;       /* days since January 1 - [0, 365]       */
  int    tm_isdst;      /* Daylight Saving Time flag             */
};
```

Time manipulation

`clock`	Returns the application's best approximation to the processor time used by the program since it was started. This low-level routine is not implemented because it strongly depends on the hardware. To determine the time in seconds, the result of clock should be divided by the value defined by `CLOCKS_PER_SEC`.
`difftime(t1,t0)`	Returns the difference *t1-t0* in seconds.
`mktime(tm *tp)`	Converts the broken-down time in the structure pointed to by *tp*, to a value of type `time_t`. The return value has the same encoding as the return value of the `time` function.
`time(*timer)`	Returns the current calendar time. This value is also assigned to `*timer`.

Time conversion

`asctime(tm *tp)`	Converts the broken-down time in the structure pointed to by *tp* into a string in the form `Mon Feb 04 16:15:14 2013\n\0`. Returns a pointer to this string.
`ctime(*timer)`	Converts the calender time pointed to by *timer* to local time in the form of a string. This is equivalent to: `asctime(localtime(timer))`
`gmtime(*timer)`	Converts the calender time pointed to by *timer* to the broken-down time, expressed as UTC. Returns a pointer to the broken-down time.
`localtime(*timer)`	Converts the calendar time pointed to by *timer* to the broken-down time, expressed as local time. Returns a pointer to the broken-down time.

Formatted time

The next function has a parallel function defined in `wchar.h`:

time.h	wchar.h
`strftime(*s,`*smax*`,*fmt,tm *`*tp*`)` `wcsftime(*s,`*smax*`,*fmt,tm *`*tp*`)`	

Formats date and time information from `struct tm` *tp* into *s* according to the specified format *fmt*. No more than *smax* characters are placed into *s*. The formatting of `strftime` is locale-specific using the `LC_TIME` category (see Section 13.1.15, *locale.h*).

You can use the next conversion specifiers:

%a   abbreviated weekday name

%A   full weekday name

%b   abbreviated month name

%B   full month name

%c   locale-specific date and time representation (same as `%a %b %e %T %Y`)

%C   last two digits of the year

%d   day of the month (01-31)

%D   same as `%m/%d/%y`

%e   day of the month (1-31), with single digits preceded by a space

%F   ISO 8601 date format: `%Y-%m-%d`

%g   last two digits of the week based year (00-99)

%G   week based year (0000–9999)

%h   same as %b

%H   hour, 24-hour clock (00-23)

%I   hour, 12-hour clock (01-12)

%j   day of the year (001-366)

%m   month (01-12)

%M   minute (00-59)

%n   replaced by newline character

%p   locale's equivalent of AM or PM

%r   locale's 12-hour clock time; same as `%I:%M:%S %p`

%R   same as `%H:%M`

%S   second (00-59)

%t   replaced by horizontal tab character

%T   ISO 8601 time format: `%H:%M:%S`

%u   ISO 8601 weekday number (1-7), Monday as first day of the week

%U   week number of the year (00-53), week 1 has the first Sunday

%V ISO 8601 week number (01-53) in the week-based year

%w weekday (0-6, Sunday is 0)

%W week number of the year (00-53), week 1 has the first Monday

%x local date representation

%X local time representation

%y year without century (00-99)

%Y year with century

%z ISO 8601 offset of time zone from UTC, or nothing

%Z time zone name, if any

%% %

13.1.28. unistd.h

The file `unistd.h` contains standard UNIX I/O functions. These functions are all implemented using file system simulation. Except for `lstat` and `fstat` which are not implemented. This header file is not defined in ISO C99.

`access(*name,mode)`	Use file system simulation to check the permissions of a file on the host. *mode* specifies the type of access and is a bit pattern constructed by a logical OR of the following values:

> `R_OK` Checks read permission.
> `W_OK` Checks write permission.
> `X_OK` Checks execute (search) permission.
> `F_OK` Checks to see if the file exists.

(FSS implementation)

`chdir(*path)`	Use file system simulation to change the current directory on the host to the directory indicated by *path*. (*FSS implementation*)
`close(fd)`	File close function. The given file descriptor should be properly closed. This function calls `_close()`. (*FSS implementation*)
`getcwd(*buf,size)`	Use file system simulation to retrieve the current directory on the host. Returns the directory name. (*FSS implementation*)
`lseek(fd,offset,whence)`	Moves read-write file offset. Calls `_lseek()`. (*FSS implementation*)
`read(fd,*buff,cnt)`	Reads a sequence of characters from a file. This function calls `_read()`. (*FSS implementation*)
`stat(*name,*buff)`	Use file system simulation to stat() a file on the host platform. (*FSS implementation*)
`lstat(*name,*buff)`	This function is identical to stat(), except in the case of a symbolic link, where the link itself is 'stat'-ed, not the file that it refers to. (*Not implemented*)
`fstat(fd,*buff)`	This function is identical to stat(), except that it uses a file descriptor instead of a name. (*Not implemented*)

`unlink(*`*name*`)` Removes the named file, so that a subsequent attempt to open it fails. (*FSS implementation)*

`write(`*fd*`,*`*buff*`,`*cnt*`)` Write a sequence of characters to a file. Calls `_write()`. (*FSS implementation)*

13.1.29. wchar.h

Many functions in `wchar.h` represent the wide-character variant of other functions so these are discussed together. (See Section 13.1.24, *stdio.h and wchar.h*, Section 13.1.25, *stdlib.h and wchar.h*, Section 13.1.26, *string.h and wchar.h* and Section 13.1.27, *time.h and wchar.h*).

The remaining functions are described below. They perform conversions between multi-byte characters and wide characters. In these functions, *ps* points to struct `mbstate_t` which holds the conversion state information necessary to convert between sequences of multibyte characters and wide characters:

```
typedef struct
{
    wchar_t         wc_value;  /* wide character value solved
                                    so far */
    unsigned short  n_bytes;   /* number of bytes of solved
                                    multibyte */
    unsigned short  encoding;  /* encoding rule for wide
                                    character <=> multibyte
                                    conversion */
} mbstate_t;
```

When multibyte characters larger than 1 byte are used, this struct will be used to store the conversion information when not all the bytes of a particular multibyte character have been read from the source. In this implementation, multi-byte characters are 1 byte long (MB_CUR_MAX and MB_LEN_MAX are defined as 1) and this will never occur.

`mbsinit(*`*ps*`)` Determines whether the object pointed to by *ps*, is an initial conversion state. Returns a non-zero value if so.

`mbsrtowcs(*`*pwcs*`,**`*src*`,`*n*`,*`*ps*`)` Restartable version of `mbstowcs`. See Section 13.1.25, *stdlib.h and wchar.h*. The initial conversion state is specified by *ps*. The input sequence of multibyte characters is specified indirectly by *src*.

`wcsrtombs(*`*s*`,**`*src*`,`*n*`,*`*ps*`)` Restartable version of `wcstombs`. See Section 13.1.25, *stdlib.h and wchar.h*. The initial conversion state is specified by *ps*. The input wide string is specified indirectly by *src*.

`mbrtowc(*`*pwc*`,*`*s*`,`*n*`,*`*ps*`)` Converts a multibyte character **s* to a wide character **pwc* according to conversion state *ps*. See also `mbtowc` in Section 13.1.25, *stdlib.h and wchar.h*.

`wcrtomb(*`*s*`,`*wc*`,*`*ps*`)` Converts a wide character wc to a multi-byte character according to conversion state *ps* and stores the multi-byte character in **s*.

`btowc(c)` Returns the wide character corresponding to character *c*. Returns WEOF on error.

`wctob(c)`	Returns the multi-byte character corresponding to the wide character *c*. The returned multi-byte character is represented as one byte. Returns EOF on error.
`mbrlen(*s,n,*ps)`	Inspects up to *n* bytes from the string **s* to see if those characters represent valid multibyte characters, relative to the conversion state held in **ps*.

13.1.30. wctype.h

Most functions in `wctype.h` represent the wide-character variant of functions declared in `ctype.h` and are discussed in Section 13.1.4, *ctype.h and wctype.h*. In addition, this header file provides extensible, locale specific functions and wide character classification.

`wctype(*property)`	Constructs a value of type `wctype_t` that describes a class of wide characters identified by the string **property*. If property identifies a valid class of wide characters according to the LC_TYPE category (see Section 13.1.15, *locale.h*) of the current locale, a non-zero value is returned that can be used as an argument in the `iswctype` function.
`iswctype(wc,desc)`	Tests whether the wide character *wc* is a member of the class represented by `wctype_t` *desc*. Returns a non-zero value if tested true.

Function	Equivalent to locale specific test
`iswalnum(wc)`	`iswctype(wc,wctype("alnum"))`
`iswalpha(wc)`	`iswctype(wc,wctype("alpha"))`
`iswcntrl(wc)`	`iswctype(wc,wctype("cntrl"))`
`iswdigit(wc)`	`iswctype(wc,wctype("digit"))`
`iswgraph(wc)`	`iswctype(wc,wctype("graph"))`
`iswlower(wc)`	`iswctype(wc,wctype("lower"))`
`iswprint(wc)`	`iswctype(wc,wctype("print"))`
`iswpunct(wc)`	`iswctype(wc,wctype("punct"))`
`iswspace(wc)`	`iswctype(wc,wctype("space"))`
`iswupper(wc)`	`iswctype(wc,wctype("upper"))`
`iswxditig(wc)`	`iswctype(wc,wctype("xdigit"))`

`wctrans(*property)`	Constructs a value of type `wctype_t` that describes a mapping between wide characters identified by the string **property*. If property identifies a valid mapping of wide characters according to the LC_TYPE category (see Section 13.1.15, *locale.h*) of the current locale, a non-zero value is returned that can be used as an argument in the `towctrans` function.
`towctrans(wc,desc)`	Transforms wide character *wc* into another wide-character, described by *desc*.

Function	Equivalent to locale specific transformation
`towlower(wc)`	`towctrans(wc,wctrans("tolower")`

Function	Equivalent to locale specific transformation
`towupper(`*`wc`*`)`	`towctrans(`*`wc`*`,wctrans("toupper")`

13.2. C Library Reentrancy

Some of the functions in the C library are reentrant, others are not. The table below shows the functions in the C library, and whether they are reentrant or not. A dash means that the function is reentrant. Note that some of the functions are not reentrant because they set the global variable 'errno' (or call other functions that eventually set 'errno'). If your program does not check this variable and errno is the only reason for the function not being reentrant, these functions can be assumed reentrant as well.

The explanation of the cause why a function is not reentrant sometimes refers to a footnote because the explanation is too lengthy for the table.

Function	Not reentrant because
`_close`	Uses global File System Simulation buffer, _dbg_request
`_doflt`	Uses I/O functions which modify iob[ ]. See (1).
`_doprint`	Uses indirect access to static iob[ ] array. See (1).
`_doscan`	Uses indirect access to iob[ ] and calls ungetc (access to local static ungetc[ ] buffer). See (1).
`_Exit`	See exit.
`_filbuf`	Uses iob[ ], which is not reentrant. See (1).
`_flsbuf`	Uses iob[ ]. See (1).
`_getflt`	Uses iob[ ]. See (1).
`_iob`	Defines static iob[ ]. See (1).
`_lseek`	Uses global File System Simulation buffer, _dbg_request
`_open`	Uses global File System Simulation buffer, _dbg_request
`_read`	Uses global File System Simulation buffer, _dbg_request
`_unlink`	Uses global File System Simulation buffer, _dbg_request
`_write`	Uses global File System Simulation buffer, _dbg_request
`abort`	Calls exit
`abs labs llabs`	-
`access`	Uses global File System Simulation buffer, _dbg_request
`acos acosf acosl`	Sets errno.
`acosh acoshf acoshl`	Sets errno via calls to other functions.
`asctime`	asctime defines static array for broken-down time string.
`asin asinf asinl`	Sets errno.
`asinh asinhf asinhl`	Sets errno via calls to other functions.
`atan atanf atanl`	-

Function	Not reentrant because
`atan2 atan2f atan2l`	-
`atanh atanhf atanhl`	Sets errno via calls to other functions.
`atexit`	atexit defines static array with function pointers to execute at exit of program.
`atof`	-
`atoi`	-
`atol`	-
`bsearch`	-
`btowc`	-
`cabs cabsf cabsl`	Sets errno via calls to other functions.
`cacos cacosf cacosl`	Sets errno via calls to other functions.
`cacosh cacosh cfacoshl`	Sets errno via calls to other functions.
`calloc`	calloc uses static buffer management structures. See malloc (5).
`carg cargf cargl`	-
`casin casinf casinl`	Sets errno via calls to other functions.
`casinh casinh cfasinhl`	Sets errno via calls to other functions.
`catan catanf catanl`	Sets errno via calls to other functions.
`catanh catanhf catanhl`	Sets errno via calls to other functions.
`cbrt cbrtf cbrtl`	-
`ccos ccosf ccosl`	Sets errno via calls to other functions.
`ccosh ccoshf ccoshl`	Sets errno via calls to other functions.
`ceil ceilf ceill`	-
`cexp cexpf cexpl`	Sets errno via calls to other functions.
`chdir`	Uses global File System Simulation buffer, _dbg_request
`cimag cimagf cimagl`	-
`cleanup`	Calls fclose. See (1)
`clearerr`	Modifies iob[ ]. See (1)
`clock`	Uses global File System Simulation buffer, _dbg_request
`clog clogf clogl`	Sets errno via calls to other functions.
`close`	Calls _close
`conj conjf conjl`	-
`copysign copysignf copysignl`	-
`cos cosf cosl`	-
`cosh coshf coshl`	cosh calls exp(), which sets errno. If errno is discarded, cosh is reentrant.

Function	Not reentrant because
cpow cpowf cpowl	Sets errno via calls to other functions.
cproj cprojf cprojl	-
creal crealf creall	-
csin csinf csinl	Sets errno via calls to other functions.
csinh csinhf csinhl	Sets errno via calls to other functions.
csqrt csqrtf csqrtl	Sets errno via calls to other functions.
ctan ctanf ctanl	Sets errno via calls to other functions.
ctanh ctanhf ctanhl	Sets errno via calls to other functions.
ctime	Calls asctime
difftime	-
div ldiv lldiv	-
erf erfl erff	-
erfc erfcf erfcl	-
exit	Calls fclose indirectly which uses iob[ ] calls functions in _atexit array. See (1). To make exit reentrant kernel support is required.
exp expf expl	Sets errno.
exp2 exp2f exp2l	Sets errno.
expm1 expm1f expm1l	Sets errno via calls to other functions.
fabs fabsf fabsl	-
fclose	Uses values in iob[ ]. See (1).
fdim fdimf fdiml	-
feclearexcept	Writes PSW bits.
fegetenv	- (reads PSW bits)
fegetexceptflag	- (reads PSW bits via calls to other functions)
fegetround	- (reads PSW bits)
feholdexept	Reads/writes PSW bits via calls to other functions.
feof	Uses values in iob[ ]. See (1).
feraiseexcept	Writes PSW bits.
ferror	Uses values in iob[ ]. See (1).
fesetenv	Writes PSW bits.
fesetexceptflag	Writes PSW bits via calls to other functions.
fesetround	Writes PSW bits.
fetestexcept	- (reads PSW bits)
feupdateenv	Writes PSW bits via calls to other functions.
fflush	Modifies iob[ ]. See (1).

Function	Not reentrant because
fgetc fgetwc	Uses pointer to iob[ ]. See (1).
fgetpos	Sets the variable errno and uses pointer to iob[ ]. See (1) / (2).
fgets fgetws	Uses iob[ ]. See (1).
floor floorf floorl	-
fma fmaf fmal	-
fmax fmaxf fmaxl	-
fmin fminf fminl	-
fmod fmodf fmodl	-
fopen	Uses iob[ ] and calls malloc when file open for buffered IO. See (1)
fpclassify	-
fprintf fwprintf	Uses iob[ ]. See (1).
fputc fputwc	Uses iob[ ]. See (1).
fputs fputws	Uses iob[ ]. See (1).
fread	Calls fgetc. See (1).
free	free uses static buffer management structures. See malloc (5).
freopen	Modifies iob[ ]. See (1).
frexp frexpf frexpl	-
fscanf fwscanf	Uses iob[ ]. See (1)
fseek	Uses iob[ ] and calls _lseek. Accesses ungetc[ ] array. See (1).
fsetpos	Uses iob[ ] and sets errno. See (1) / (2).
fstat	(*Not implemented*)
ftell	Uses iob[ ] and sets errno. Calls _lseek. See (1) / (2).
fwrite	Uses iob[ ]. See (1).
getc getwc	Uses iob[ ]. See (1).
getchar getwchar	Uses iob[ ]. See (1).
getcwd	Uses global File System Simulation buffer, _dbg_request
getenv	Skeleton only.
gets getws	Uses iob[ ]. See (1).
gmtime	gmtime defines static structure
hypot hypotf hypotl	Sets errno via calls to other functions.
ilogb ilogbf ilogbl	Sets errno.
imaxabs	-
imaxdiv	-
isalnum iswalnum	-
isalpha iswalpha	-

Function	Not reentrant because
`isascii iswascii`	-
`iscntrl iswcntrl`	-
`isdigit iswdigit`	-
`isfinite`	-
`isgraph iswgraph`	-
`isgreater`	-
`isgreaterequal`	-
`isinf`	-
`isless`	-
`islessequal`	-
`islessgreater`	-
`islower iswlower`	-
`isnan`	-
`isnormal`	-
`isprint iswprint`	-
`ispunct iswpunct`	-
`isspace iswspace`	-
`isunordered`	-
`isupper iswupper`	-
`iswalnum`	-
`iswalpha`	-
`iswcntrl`	-
`iswctype`	-
`iswdigit`	-
`iswgraph`	-
`iswlower`	-
`iswprint`	-
`iswpunct`	-
`iswspace`	-
`iswupper`	-
`iswxditig`	-
`isxdigit iswxdigit`	-
`ldexp ldexpf ldexpl`	Sets errno. See (2).
`lgamma lgammaf lgammal`	Sets errno.
`llrint llrintf llrintl`	-

Function	Not reentrant because
`llround llroundf llroundl`	Sets errno.
`localeconv`	N.A.; skeleton function
`localtime`	-
`log logf logl`	Sets errno. See (2).
`log10 log10f log10l`	Sets errno via calls to other functions.
`log1p log1pf log1pl`	Sets errno.
`log2 log2f log2l`	Sets errno.
`logb logbf logbl`	Sets errno.
`longjmp`	-
`lrint lrintf lrintl`	-
`lround lroundf lroundl`	Sets errno.
`lseek`	Calls _lseek
`lstat`	(*Not implemented*)
`malloc`	Needs kernel support. See (5).
`mblen`	N.A., skeleton function
`mbrlen`	Sets errno.
`mbrtowc`	Sets errno.
`mbsinit`	-
`mbsrtowcs`	Sets errno.
`mbstowcs`	N.A., skeleton function
`mbtowc`	N.A., skeleton function
`memchr wmemchr`	-
`memcmp wmemcmp`	-
`memcpy wmemcpy`	-
`memmove wmemmove`	-
`memset wmemset`	-
`mktime`	-
`modf modff modfl`	-
`nan nanf nanl`	-
`nearbyint nearbyintf nearbyintl`	-
`nextafter nextafterf nextafterl`	-
`nexttoward nexttowardf nexttowardl`	-
`offsetof`	-

Function	Not reentrant because
open	Calls _open
perror	Uses errno. See (2)
pow powf powl	Sets errno. See (2)
printf wprintf	Uses iob[ ]. See (1)
putc putwc	Uses iob[ ]. See (1)
putchar putwchar	Uses iob[ ]. See (1)
puts	Uses iob[ ]. See (1)
qsort	-
raise	Updates the signal handler table
rand	Uses static variable to remember latest random number. Must diverge from ISO C standard to define reentrant rand. See (4).
read	Calls _read
realloc	See malloc (5).
remainder remainderf remainderl	-
remove	Uses global File System Simulation buffer, _dbg_request
remquo remquof remquol	-
rename	Uses global File System Simulation buffer, _dbg_request
rewind	Eventually calls _lseek
rint rintf rintl	-
round roundf roundl	-
scalbln scalblnf scalblnl	-
scalbn scalbnf scalbnl	-
scanf wscanf	Uses iob[ ], calls _doscan. See (1).
setbuf	Sets iob[ ]. See (1).
setjmp	-
setlocale	N.A.; skeleton function
setvbuf	Sets iob and calls malloc. See (1) / (5).
signal	Updates the signal handler table
signbit	-
sin sinf sinl	-
sinh sinhf sinhl	Sets errno via calls to other functions.
snprintf swprintf	Sets errno. See (2).
sprintf	Sets errno. See (2).
sqrt sqrtf sqrtl	Sets errno. See (2).
srand	See rand

Function	Not reentrant because
sscanf swscanf	Sets errno via calls to other functions.
stat	Uses global File System Simulation buffer, _dbg_request
strcat wcscat	-
strchr wcschr	-
strcmp wcscmp	-
strcoll wcscoll	-
strcpy wcscpy	-
strcspn wcscspn	-
strerror	-
strftime wcsftime	-
strlen wcslen	-
strncat wcsncat	-
strncmp wcsncmp	-
strncpy wcsncpy	-
strpbrk wcspbrk	-
strrchr wcsrchr	-
strspn wcsspn	-
strstr wcsstr	-
strtod wcstod	-
strtof wcstof	-
strtoimax	Sets errno via calls to other functions.
strtok wcstok	strtok saves last position in string in local static variable. This function is not reentrant by design. See (4).
strtol wcstol	Sets errno. See (2).
strtold wcstold	-
strtoul wcstoul	Sets errno. See (2).
strtoull wcstoull	Sets errno. See (2).
strtoumax	Sets errno via calls to other functions.
strxfrm wcsxfrm	-
system	N.A; skeleton function
tan tanf tanl	Sets errno. See (2).
tanh tanhf tanhl	Sets errno via call to other functions.
tgamma tgammaf tgammal	Sets errno.
time	Uses static variable which defines initial start time
tmpfile	Uses iob[ ]. See (1).

Function	Not reentrant because
`tmpnam`	Uses local buffer to build filename. Function can be adapted to use user buffer. This makes the function non ISO C. See (4).
`toascii`	-
`tolower`	-
`toupper`	-
`towctrans`	-
`towlower`	-
`towupper`	-
`trunc truncf truncl`	-
`ungetc ungetwc`	Uses static buffer to hold unget characters for each file. Can be moved into iob structure. See (1).
`unlink`	Uses global File System Simulation buffer, _dbg_request
`vfprintf vfwprintf`	Uses iob[ ]. See (1).
`vfscanf vfwscanf`	Calls _doscan
`vprintf vwprintf`	Uses iob[ ]. See (1).
`vscanf vwscanf`	Calls _doscan
`vsprintf vswprintf`	Sets errno.
`vsscanf vswscanf`	Sets errno.
`wcrtomb`	Sets errno.
`wcsrtombs`	Sets errno.
`wcstoimax`	Sets errno via calls to other functions.
`wcstombs`	N.A.; skeleton function
`wcstoumax`	Sets errno via calls to other functions.
`wctob`	-
`wctomb`	N.A.; skeleton function
`wctrans`	-
`wctype`	-
`write`	Calls _write

Table: C library reentrancy

Several functions in the C library are not reentrant due to the following reasons:

• The `iob[]` structure is static. This influences all I/O functions.

• The `ungetc[]` array is static. This array holds the characters (one for each stream) when `ungetc()` is called.

• The variable `errno` is globally defined. Numerous functions read or modify `errno`

- `_doprint` and `_doscan` use static variables for e.g. character counting in strings.

- Some string functions use locally defined (static) buffers. This is prescribed by ANSI.

- `malloc` uses a static heap space.

The following description discusses these items in more detail. The numbers at the beginning of each paragraph relate to the number references in the table above.

(1) iob structures

The I/O part of the C library is not reentrant by design. This is mainly caused by the static declaration of the `iob[]` array. The functions which use elements of this array access these elements via pointers (`FILE *`).

Building a multi-process system that is created in one link-run is hard to do. The C language scoping rules for external variables make it difficult to create a private copy of the `iob[]` array. Currently, the `iob[]` array has external scope. Thus it is visible in every module involved in one link phase. If these modules comprise several tasks (processes) in a system each of which should have its private copy of `iob[]`, it is apparent that the `iob[]` declaration should be changed. This requires adaptation of the library to the multi-tasking environment. The library modules must use a process identification as an index for determining which `iob[]` array to use. Thus the library is suitable for interfacing to that kernel only.

Another approach for the `iob[]` declaration problem is to declare the array static in one of the modules which create a task. Thus there can be more than one `iob[]` array is the system without having conflicts at link time. This brings several restrictions: Only the module that holds the declaration of the static `iob[]` can use the standard file handles `stdin`, `stdout` and `stderr` (which are the first three entries in `iob[]`). Thus all I/O for these three file handles should be located in one module.

(2) errno declaration

Several functions in the C library set the global variable `errno`. After completion of the function the user program may consult this variable to see if some error occurred. Since most of the functions that set `errno` already have a return type (this is the reason for using `errno`) it is not possible to check successful completion via the return type.

The library routines can set `errno` to the values defined in `errno.h`. See the file `errno.h` for more information.

`errno` can be set to ERR_FORMAT by the print and scan functions in the C library if you specify illegal format strings.

`errno` will never be set to ERR_NOLONG or ERR_NOPOINT since the C library supports long and pointer conversion routines for input and output.

`errno` can be set to ERANGE by the following functions: `exp()`, `strtol()`, `strtoul()` and `tan()`. These functions may produce results that are out of the valid range for the return type. If so, the result of the function will be the largest representable value for that type and `errno` is set to ERANGE.

`errno` is set to EDOM by the following functions: `acos()`, `asin()`, `log()`, `pow()` and `sqrt()`. If the arguments for these functions are out of their valid range (e.g. `sqrt( -1 )`), `errno` is set to EDOM.

errno can be set to ERR_POS by the file positioning functions ftell(), fsetpos() and fgetpos().

(3) ungetc

Currently the ungetc buffer is static. For each file entry in the iob[] structure array, there is one character available in the buffer to unget a character.

(4) local buffers

tmpnam() creates a temporary filename and returns a pointer to a local static buffer. This is according to the ANSI definition. Changing this function such that it creates the name in a user specified buffer requires another calling interface. Thus the function would be no longer portable.

strtok() scans through a string and remembers that the string and the position in the string for subsequent calls. This function is not reentrant by design. Making it reentrant requires support of a kernel to store the information on a per process basis.

rand() generates a sequence of random numbers. The function uses the value returned by a previous call to generate the next value in the sequence. This function can be made reentrant by specifying the previous random value as one of the arguments. However, then it is no longer a standard function.

(5) malloc

Malloc uses a heap space which is assigned at locate time. Thus this implementation is not reentrant. Making a reentrant malloc requires some sort of system call to obtain free memory space on a per process basis. This is not easy to solve within the current context of the library. This requires adaptation to a kernel.

This paragraph on reentrancy applies to multi-process environments only. If reentrancy is required for calling library functions from an exception handler, another approach is required. For such a situation it is of no use to allocate e.g. multiple iob[] structures. In such a situation several pieces of code in the library have to be declared 'atomic': this means that interrupts have to be disabled while executing an atomic piece of code.

The default exception handler (exception_handler) delivered in the floating-point run-time library in except.c is not reentrant, because it uses global variables for the custom handler and status.

Chapter 14. List File Formats

This chapter describes the format of the assembler list file and the linker map file.

14.1. Assembler List File Format

The assembler list file is an additional output file of the assembler that contains information about the generated code. For details on how to generate a list file, see Section 6.5, *Generating a List File*.

The list file consists of a page header and a source listing.

Page header

The page header is repeated on every page:

```
TASKING VX-toolset for TriCore: assembler vx.yrz Build nnn SN 00000000
Title                                                           Page 1

ADDR CODE      CYCLES  LINE SOURCE LINE
```

The first line contains version information. The second line can contain a title which you can specify with the assembler control $TITLE and always contains a page number. The third line is empty and the fourth line contains the headings of the columns for the source listing.

With the assembler controls $LIST ON/OFF, $PAGE, and with the assembler option **--list-format** you can format the list file.

Source listing

The following is a sample part of a listing. An explanation of the different columns follows below.

```
ADDR CODE      CYCLES  LINE SOURCE LINE
                        1           ; Module start
                        .
                        .
0002 85rFrrrr 1    2    27          ld.a    a15,world
0006 F4AF     1    3    28          st16.a  [a10],a15
0008 91r0rr4r 1    4    29          movh.a  a4,#@his(_2_ini)
000C D944rrrr 1    5    30          lea     a4,[a4]@los(_2_ini)
0010 1Drrrrrr 1    6    31          j       printf
                        .
                        .
0000                    44 buf:     .space  4
   |   RESERVED
0003
```

ADDR	This column contains the memory address. The address is a hexadecimal number that represents the offset from the beginning of a relocatable section or the absolute address for an absolute section. The address only appears on lines that generate object code.
CODE	This is the object code generated by the assembler for this source line, displayed in hexadecimal format. The displayed code need not be the same as the generated code that is entered in the object module. The code can also be relocatable code. In this case the letter '**r**' is printed for the relocatable code part in the listing. For lines that allocate space, the code field contains the text "RESERVED". For lines that initialize a buffer, the code field lists one value followed by the word "REPEATS".
CYCLES	The first number in this column is the number of instruction cycles needed to execute the instruction(s) as generated in the CODE field. The second number is the accumulated cycle count of this section.
LINE	This column contains the line number. This is a decimal number indicating each input line, starting from 1 and incrementing with each source line.
SOURCE LINE	This column contains the source text. This is a copy of the source line from the assembly source file.

For the `.SET` and `.EQU` directives the `ADDR` and `CODE` columns do not apply. The symbol value is listed instead.

14.2. Linker Map File Format

The linker map file is an additional output file of the linker that shows how the linker has mapped the sections and symbols from the various object files (`.o`) to output sections. The locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address. For details on how to generate a map file, see Section 7.10, *Generating a Map File*.

With the linker option **--map-file-format** you can specify which parts of the map file you want to see.

In Eclipse the linker map file (*project*.`mapxml`) is generated in the output directory of the build configuration, usually `Debug` or `Release`. You can open the map file by double-clicking on the file name.

Each page displays a part of the map file. You can use the drop-down list or the Outline view to navigate through the different tables and you can use the following buttons.

Icon	Action	Description
⇦	Back	Goes back one page in the history list.
⇨	Forward	Goes forward one page in the history list.
🗔	Next Table	Shows the next table from the drop-down list.
🗔	Previous Table	Shows the previous table from the drop-down list.

When you right-click in the view, a popup menu appears (for example, to reset the layout of a table). The meaning of the different parts is:

Tool and Invocation

This part of the map file contains information about the linker, its version header information, binary location and which options are used to call it.

Used Resources

This part of the map file shows the memory usage at memory level and space level. The largest free block of memory (`Largest gap`) is also shown. This part also contains an estimation of the stack usage.

Explanation of the columns:

Memory	The names of the memory as defined in the linker script file (`*.lsl`).
Code	The size of all executable sections.
Data	The size of all non-executable sections (not including stacks, heaps, debug sections in non-alloc space).

Reserved	The total size of reserved memories, reserved ranges, reserved special sections, stacks, heaps, alignment protections, sections located in non-alloc space (debug sections). In fact, this size is the same as the size in the Total column minus the size of all other columns.
Free	The free memory area addressable by this core. This area is accessible for unrestricted items.
Total	The total memory area addressable by this core.
Space	The names of the address spaces as defined in the linker script file (`*.lsl`). The names are constructed of the `derivative` name followed by a colon ':', the `core` name, another colon ':' and the `space` name. For example: `spe:tc:linear`.
Native used ...	The size of sections located in this space.
Foreign used	The size of all sections destined for/located in other spaces, but because of overlap in spaces consume memory in this space.
Stack Name	The name(s) of the stack(s) as defined in the linker script file (`*.lsl`).
Used	An estimation of the stack usage. The linker calculates the required stack size by using information (`.CALLS` directives) generated by the compiler. If for example recursion is detected, the calculated stack size is inaccurate, therefore this is an estimation only. The calculated stack size is supposed to be smaller than the actual allocated stack size. If that is not the case, then a warning is given.

Processed Files

This part of the map file shows all processed files. This also includes object files that are extracted from a library, with the symbol that led to the extraction.

Link Result

This part of the map file shows per object file how the link phase has mapped the sections from the various object files (`.o`) to output sections.

[in] File	The name of an input object file.
[in] Section	A section name and id from the input object file. The number between '( )' uniquely identifies the section.
[in] Size	The size of the input section.
[out] Offset	The offset relative to the start of the output section.
[out] Section	The resulting output section name and id.
[out] Size	The size of the output section.

Module Local Symbols

This part of the map file shows a table for each local scope within an object file. Each table has three columns, 1 the symbol name, 2 the address of the symbol and 3 the space where the symbol resides in. The table is sorted on symbol name within each space.

By default this part is not shown in the map file. You have to turn this part on manually with linker option **--map-file-format=+statics** (module local symbols).

Cross References

This part of the map file lists all symbols defined in the object modules and for each symbol the object modules that contain a reference to the symbol are shown. Also, symbols that remain undefined are shown.

Call Graph

This part of the map file contains a schematic overview that shows how (library) functions call each other. To obtain call graph information, the assembly file must contain .CALLS directives.

The following example is a part of a call graph in the textual version of the map file (.map):

```
_START [ustack_tc0:0,24]
|
+-- cstart.src:__init_sp [ustack_tc0:0,24]
    |
    +-- cstart.src:_start [ustack_tc0:0,24]
        |
        +-- cstart.src:.cocofun_1 [ustack_tc0:0,0]
        |
        +-- cstart.src:.cocofun_2 [ustack_tc0:0,0]
        |
        +-- cstart.src:.cocofun_5 [ustack_tc0:0,0]
        |
        +-- cstart.src:.cocofun_3 [ustack_tc0:0,0]
        |
        +-- cstart.src:.cocofun_4 [ustack_tc0:0,0]
        |
        +-- main *
        |
        +-- _c_init [ustack_tc0:0,0]
        |   |
        |   +-- _c_init_entry *
        |
        +-- exit *

* main [ustack_tc0:0,24]
|
+-- printf [ustack_tc0:16,24]
    |
    +-- _doprint [ustack_tc0:0,8]
        |
        +-- _io_putc [ustack_tc0:0,8]
            |
            +-- fputc [ustack_tc0:0,8]
                |
```

```
            +-- _flsbuf [ustack_tc0:8,8]
                |
            +-- _fflush *
                |
            +-- _write *
```

- A * after a function name indicates a caller, which calls a function which is listed separately in the call graph.

- A * in front of a function name indicates a callee. This function is referenced by a caller.

- An additional R (not shown in this example) indicates this function is part of a recursive call chain. If both a leaf and the root of a tree are marked this way, all nodes in between are in a recursive chain.

- [ ] after a function contains information about the stack usage. The first field is the name of the stack, followed by the amount of stack used by the function and the amount of stack used by the function including its callees.

In the graphical version of the map file, you can click the + or - sign to expand or collapse a single node. Use the ⊞ / ⊟ buttons to expand/collapse all nodes in the call graph. Hover the mouse over a function (root, callee or node) to see information about the stack usage.

Icon	Meaning	Description
⚛	Root	This function is the top of the call graph. If there are interrupt handlers, there can be several roots.
≋ ≋	Callee	This function is referenced by several caller functions. Right-click on the function and select **Expand all References** to see all functions that reference this function. Select **Back to Caller** to return to the calling function. ↻ indicates a recursive function.
●	Node	A normal node (function) in the call graph.

Icon	Meaning	Description
➡ ➡	Caller	This function calls a function which is listed separately in the call graph. Right-click on the function and select **Go to Callee** to see the callee. Hover the mouse over the function to see a popup with all callees. ↻ indicates a recursive function.

Overlay

This part is empty for the TriCore.

Locate Result: Sections

This part of the map file shows the absolute position of each section in the absolute object file. It is organized per address space, memory chip and group and sorted on space address. In Eclipse, right-click in the table and select **Configure Columns** to specify which columns you want to see. If you hover the mouse over a section, you get a popup with information about the section. If you select a range of sections, in the Fast View bar (at the bottom) you will see information about the selected range, such as the total size, how many sections are selected and how many gaps are present.

#	The line number and default sort order.
Section **Section name** **Section number**	The name and id of the section. The number between '( )' uniquely identifies the section. Sections within square brackets **[ ]** will be copied during initialization from ROM to the corresponding section name in RAM.
Sect. size (hex) **Sect. size (dec)**	The size of the section in minimum addressable units (hexadecimal or decimal).
Group	Sections can be ordered in groups. These are the names of the groups as defined in the linker script file (*.lsl) with the keyword group in the section_layout definition. The name that is displayed is the name of the deepest nested group.
Start address	The first address of the section in the address space.
End address	The last address of the section in the address space.
Symbols in sect.	The names of the external symbols that are referenced in the section. See **Locate Result: Symbols** below.
Defined in	The names of the input modules the section is defined in. See **Link Result: [in] File** above.
Referenced in	The names of the modules that contain a reference to the section. See **Cross References** above.
Address space	The name of the address space.
Chip name	The names of the memory chips as defined in the linker script file (*.lsl) in the memory definitions.
Chip addr	The absolute offset of the section from the start of a memory chip.
Locate type:properties	The locate rule type and properties. See **Locate Rules** below.

The following buttons are available in this part of the map file.

Icon	Action	Description
▦	Configure Section Filter	Opens the Configure Section Filter dialog. Here you can select which sections you want to see in the map file and how.
🖌	Enable Highlighting	All sections that are marked with "Highlight" in the Configure Section Filter dialog will be highlighted in the table.
�	Enable Collapsing	All sections that are marked with "Collapse" in the Configure Section Filter dialog will appear collapsed in the table.
⇥	Only Show Matching Lines	All lines that are not part of the selection in the Configure Section Filter dialog will be hidden.
⸝	Show Gaps	Also shows the gaps in the map file. Click the button again to hide the gaps.

Configure Section Filter Dialog

In this dialog you can filter which sections you want to see in the map file and how. Click **Add** to add a new filter. Explanation of the columns and fields:

Highlight   Marks the section as a candidate for highlighting. Turn on **Enable Highlighting** to see the effect.

Color   The highlight color.

Collapse   Marks the section as a candidate for collapsing. Turn on **Enable Collapsing** to see the effect.

Section name   A filter to select a section or group of sections. Wildcards are allowed. Wildcards follow the rules of regular expressions. To get help on which wildcards are supported, press **Ctrl-space**. Click an item in the list for help, double-click to add the wildcard.

Start address   The first address of the section in the address space for this filter.

End address   The last address of the section in the address space for this filter.

Address space   The name of the address space.

Chip name   The name of the memory chip as defined in the linker script file (`*.lsl`) in the `memory` definitions.

Hide gaps smaller than   If gaps are shown in the map file, here you can limit the number of gaps you want to see.

The meaning of the check boxes is the same as the corresponding buttons available in this part of the map file.

Locate Result: Symbols

This part of the map file lists all external symbols per address space name.

Address   The absolute address of the symbol in the address space.

Name   The name of the symbol.

Space	The names of the address spaces as defined in the linker script file (`*.lsl`). The names are constructed of the `derivative` name followed by a colon ':', the `core` name, another colon ':' and the `space` name. For example: `spe:tc:linear`.

Processor and Memory

This part of the map file shows the processor and memory information of the linker script file.

By default this part is not shown in the map file. You have to turn this part on manually with linker option **--map-file-format=+lsl** (processor and memory info). You can print this information to a separate file with linker option **--lsl-dump**.

You can click the + or - sign to expand or collapse a part of the information.

Locate Rules

This part of the map file shows the rules the linker uses to locate sections.

Address space	The names of the address spaces as defined in the linker script file (`*.lsl`). The names are constructed of the `derivative` name followed by a colon ':', the `core` name, another colon ':' and the `space` name.
Type	The rule type:

`ordered/contiguous/clustered/unrestricted`

Specifies how sections are grouped. By default, a group is 'unrestricted' which means that the linker has total freedom to place the sections of the group in the address space.

`absolute`

The section must be located at the address shown in the Properties column.

`ranged`

The section must be located anywhere in the address ranges shown in the Properties column; end addresses are not included in the range.

`page`

The sections must be located in some address range with a size not larger than shown in the Properties column; the first number is the page size, the second part is the address range restriction within the page.

`ranged page`

Both the ranged and the paged restriction apply. In the Properties column the range restriction is listed first, followed by the paged restriction between parenthesis.

`ballooned`

After locating all sections, the largest remaining gap in the space is used completely for the stack and/or heap.

Properties	The contents depends on the Type column.
Prio	The locate priority of the rule. A higher priority value gives a rule precedence over a rule with a lower priority, but only if the two rules have the same type and the same properties. The relative order of rules of different types or different properties is not affected by this priority value. You can set the priority with the `priority group attribute` in LSL
Sections	The sections to which the rule applies; restrictions between sections are shown in this column:

```
<    ordered
|    contiguous
+    clustered
```

For contiguous sections, the linker uses the section order as shown here. Clustered sections can be located in any relative order.

Removed Sections

This part of the map file shows the sections which are removed from the output file as a result of the optimization option to delete unreferenced sections and or duplicate code or constant data (linker option **--optimize=cxy**).

Section	The name of the section which has been removed.
File	The name of the input object file where the section is removed from.
Library	The name of the library where the object file is part of.
Symbol	The symbols that were present in the section.
Reason	The reason why the section has been removed. This can be because the section is unreferenced or duplicated.

Chapter 15. Object File Formats

This chapter describes the format of several object files.

15.1. ELF/DWARF Object Format

The TASKING VX-toolset for TriCore produces objects in the ELF/DWARF 3 format.

The ELF/DWARF Object Format for the TriCore toolset follows the convention as described in the *TriCore Embedded Application Binary Interface* [Infineon].

For a complete description of the ELF and DWARF formats, please refer to the *Tool Interface Standard (TIS)*.

15.2. Intel Hex Record Format

Intel Hex records describe the hexadecimal object file format for 8-bit, 16-bit and 32-bit microprocessors. The hexadecimal object file is an ASCII representation of an absolute binary object file. There are six different types of records:

- Data Record (8-, 16, or 32-bit formats)

- End of File Record (8-, 16, or 32-bit formats)

- Extended Segment Address Record (16, or 32-bit formats)

- Start Segment Address Record (16, or 32-bit formats)

- Extended Linear Address Record (32-bit format only)

- Start Linear Address Record (32-bit format only)

To generate an Intel Hex output file:

1. From the **Project** menu, select **Properties**

   The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Output Format**.

4. Enable the option **Generate Intel Hex format file**.

5. (Optional) Specify the **Size of addresses (in bytes) for Intel Hex records**.

6. (Optional) Enable or disable the option **Emit start address record**.

By default the linker generates records in the 32-bit format (4-byte addresses).

General Record Format

In the output file, the record format is:

:	*length*	*offset*	*type*	*content*	*checksum*

where:

:	is the record header.
length	is the record length which specifies the number of bytes of the *content* field. This value occupies one byte (two hexadecimal digits). The linker outputs records of 255 bytes (32 hexadecimal digits) or less; that is, *length* is never greater than 0xFF.
offset	is the starting load offset specifying an absolute address in memory where the data is to be located when loaded by a tool. This field is two bytes long. This field is only used for Data Records. In other records this field is coded as four ASCII zero characters ('0000').
type	is the record type. This value occupies one byte (two hexadecimal digits). The record types are:

Byte Type	Record Type
00	Data
01	End of file
02	Extended segment address (not used)
03	Start segment address (not used)
04	Extended linear address (32-bit)
05	Start linear address (32-bit)

content	is the information contained in the record. This depends on the record type.
checksum	is the record checksum. The linker computes the checksum by first adding the binary representation of the previous bytes (from length to content). The linker then computes the result of sum modulo 256 and subtracts the remainder from 256 (two's complement). Therefore, the sum of all bytes following the header is zero.

Extended Linear Address Record

The Extended Linear Address Record specifies the two most significant bytes (bits 16-31) of the absolute address of the first data byte in a subsequent Data Record:

:	02	0000	04	*upper_address*	*checksum*

The 32-bit absolute address of a byte in a Data Record is calculated as:

```
( address + offset + index ) modulo 4G
```

where:

address	is the base address, where the two most significant bytes are the upper_address and the two least significant bytes are zero.
offset	is the 16-bit offset from the Data Record.
index	is the index of the data byte within the Data Record (0 for the first byte).

Example:

```
:0200000400FFFB
 | |   | |     |_ checksum
 | |   | |_ upper_address
 | |   |_ type
 | |_ offset
 |_ length
```

Data Record

The Data Record specifies the actual program code and data.

:	*length*	*offset*	00	*data*	*checksum*

The *length* byte specifies the number of *data* bytes. The linker has an option (**--hex-record-size**) that controls the length of the output buffer for generating Data records. The default buffer length is 32 bytes.

The *offset* is the 16-bit starting load offset. Together with the address specified in the Extended Address Record it specifies an absolute address in memory where the data is to be located when loaded by a tool.

Example:

```
:0F00200000232222754E00754F04AF4FAE4E22C3
 | |   | |                              |_ checksum
 | |   | |_ data
 | |   |_ type
 | |_ offset
 |_ length
```

Start Linear Address Record

The Start Linear Address Record contains the 32-bit program execution start address.

:	04	0000	05	*address*	*checksum*

With linker option **--hex-format=S** you can prevent the linker from emitting this record.

Example:

```
:04000005A000000057
 | |    | |         |_ checksum
 | |    | |_ address
 | |    |_ type
 | |_ offset
 |_ length
```

End of File Record

The hexadecimal file always ends with the following end-of-file record:

```
:00000001FF
 | |    | |_ checksum
 | |    |_ type
 | |_ offset
 |_ length
```

15.3. Motorola S-Record Format

To generate a Motorola S-record output file:

1. From the **Project** menu, select **Properties**

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Output Format**.

4. Enable the option **Generate S-records file**.

5. (Optional) Specify the **Size of addresses (in bytes) for Motorola S records**.

By default, the linker produces output in Motorola S-record format with three types of S-records (4-byte addresses): S0, S3 and S7. Depending on the size of addresses you can force other types of S-records. They have the following layout:

S0 - record

S0	length	0000	comment	checksum

A linker generated S-record file starts with an S0 record with the following contents:

```
        l t c
S00600006C7463B6
```

The S0 record is a comment record and does not contain relevant information for program execution.

where:

S0	is a comment record and does not contain relevant information for program execution.
length	represents the number of bytes in the record, not including the record type and length byte. This value occupies one byte (two hexadecimal digits).
comment	contains the name of the linker.
checksum	is the record checksum. The linker computes the checksum by first adding the binary representation of the bytes following the record type (starting with the *length* byte) to just before the checksum. Then the one's complement is calculated of this sum. The least significant byte of the result is the checksum. The sum of all bytes following the record type is 0xFF.

S1 / S2 / S3 - record

This record is the program code and data record for 2-byte, 3-byte or 4-byte addresses respectively.

S1	*length*	*address*	*code bytes*	*checksum*

S2	*length*	*address*	*code bytes*	*checksum*

S3	*length*	*address*	*code bytes*	*checksum*

where:

S1	is the program code and data record for 2-byte addresses.
S2	is the program code and data record for 3-byte addresses.
S3	is the program code and data record for 4-byte addresses (this is the default).
length	represents the number of bytes in the record, not including the record type and length byte. This value occupies one byte (two hexadecimal digits).
address	contains the code or data address.
code bytes	contains the actual program code and data.
checksum	is the record checksum. The checksum calculation is identical to S0.

Example:

```
S3070000FFFE6E6825
  | |         |   |_ checksum
  | |         |_ code
  | |_ address
  |_ length
```

S7 / S8 / S9 - record

This record is the termination record for 4-byte, 3-byte or 2-byte addresses respectively.

S7	length	address	checksum

S8	length	address	checksum

S9	length	address	checksum

where:

S7	is the termination record for 4-byte addresses (this is the default). S7 is the corresponding termination record for S3 records.
S8	is the termination record for 3-byte addresses. S8 is the corresponding termination record for S2 records.
S9	is the termination record for 2-byte addresses. S9 is the corresponding termination record for S1 records.
length	represents the number of bytes in the record, not including the record type and length byte. This value occupies one byte (two hexadecimal digits).
address	contains the program start address.
checksum	is the record checksum. The checksum calculation is identical to S0.

Example:

```
S705A00000005A
  | |          |_checksum
  | |_ address
  |_ length
```

Chapter 16. Linker Script Language (LSL)

To make full use of the linker, you can write a script with information about the architecture of the target processor and locating information. The language for the script is called the *Linker Script Language (LSL)*. This chapter first describes the structure of an LSL file. The next section contains a summary of the LSL syntax. In the remaining sections, the semantics of the Linker Script Language is explained.

The TASKING linker is a target independent linker/locator that can simultaneously link and locate all programs for all cores available on a target board. The target board may be of arbitrary complexity. A simple target board may contain one standard processor with some external memory that executes one task. A complex target board may contain multiple standard processors and DSPs combined with configurable IP-cores loaded in an FPGA. Each core may execute a different program, and external memory may be shared by multiple cores.

LSL serves two purposes. First it enables you to specify the characteristics (that are of interest to the linker) of your specific target board and of the cores installed on the board. Second it enables you to specify how sections should be located in memory.

16.1. Structure of a Linker Script File

A script file consists of several definitions. The definitions can appear in any order.

The architecture definition (required)

In essence an *architecture definition* describes how the linker should convert logical addresses into physical addresses for a given type of core. If the core supports multiple address spaces, then for each space the linker must know how to perform this conversion. In this context a physical address is an offset on a given internal or external bus. Additionally the architecture definition contains information about items such as the (hardware) stack and the interrupt vector table.

This specification is normally written by Altium. Altium supplies LSL files in the `include.lsl` directory. The architecture definition of the LSL file should not be changed by you unless you also modify the core's hardware architecture. If the LSL file describes a multi-core system an architecture definition must be available for each different type of core.

See Section 16.4, *Semantics of the Architecture Definition* for detailed descriptions of LSL in the architecture definition.

The derivative definition

The *derivative definition* describes the configuration of the internal (on-chip) bus and memory system. Basically it tells the linker how to convert offsets on the buses specified in the architecture definition into offsets in internal memory. Microcontrollers and DSPs often have internal memory and I/O sub-systems apart from one or more cores. The design of such a chip is called a *derivative*.

Altium provides LSL descriptions of supported derivatives, along with "SFR files", which provide easy access to registers in I/O sub-systems from C and assembly programs. When you build an ASIC or use a derivative that is not (yet) supported by the TASKING tools, you may have to write a derivative definition.

When you want to use multiple cores of the same type, you must instantiate the cores in a derivative definition, since the linker automatically instantiates only a single core for an unused architecture.

See Section 16.5, *Semantics of the Derivative Definition* for a detailed description of LSL in the derivative definition.

The processor definition

The *processor definition* describes an instance of a derivative. Typically the processor definition instantiates one derivative only (single-core processor). A processor that contains multiple cores having the same (homogeneous) or different (heterogeneous) architecture can also be described by instantiating multiple derivatives of the same or different types in separate processor definitions.

See Section 16.6, *Semantics of the Board Specification* for a detailed description of LSL in the processor definition.

The memory and bus definitions (optional)

Memory and bus definitions are used within the context of a derivative definition to specify internal memory and on-chip buses. In the context of a board specification the memory and bus definitions are used to define external (off-chip) memory and buses. Given the above definitions the linker can convert a logical address into an offset into an on-chip or off-chip memory device.

See Section 16.6.3, *Defining External Memory and Buses*, for more information on how to specify the external physical memory layout. *Internal* memory for a processor should be defined in the derivative definition for that processor.

The board specification

The processor definition and memory and bus definitions together form a *board specification*. LSL provides language constructs to easily describe single-core and heterogeneous or homogeneous multi-core systems. The board specification describes all characteristics of your target board's system buses, memory devices, I/O sub-systems, and cores that are of interest to the linker. Based on the information provided in the board specification the linker can for each core:

- convert a logical address to an offset within a memory device

- locate sections in physical memory

- maintain an overall view of the used and free physical memory within the whole system while locating

The section layout definition (optional)

The optional section layout definition enables you to exactly control where input sections are located. Features are provided such as: the ability to place sections at a given load-address or run-time address, to place sections in a given order, and to overlay code and/or data sections.

Which object files (sections) constitute the task that will run on a given core is specified on the command line when you invoke the linker. The linker will link and locate all sections of all tasks simultaneously. From the section layout definition the linker can deduce where a given section may be located in memory,

form the board specification the linker can deduce which physical memory is (still) available while locating the section.

See Section 16.8, *Semantics of the Section Layout Definition*, for more information on how to locate a section at a specific place in memory.

Skeleton of a Linker Script File

```
architecture architecture_name
{
    // Specification core architecture
}

derivative derivative_name
{
    // Derivative definition
}

processor processor_name
{
    // Processor definition
}

memory and/or bus definitions

section_layout space_name
{
    // section placement statements
}
```

16.2. Syntax of the Linker Script Language

This section describes what the LSL language looks like. An LSL document is stored as a file coded in UTF-8 with extension .lsl. Before processing an LSL file, the linker preprocesses it using a standard C preprocessor. Following this, the linker interprets the LSL file using a scanner and parser. Finally, the linker uses the information found in the LSL file to guide the locating process.

16.2.1. Preprocessing

When the linker loads an LSL file, the linker processes it with a C-style prepocessor. As such, it strips C and C++ comments. You can use the standard ISO C preprocessor directives, such as #include, #define, #if/#ifdef/#else/#endif, #error.

For example:

```
#include "arch.lsl"
```

Preprocess and include the file arch.lsl at this point in the LSL file.

16.2.2. Lexical Syntax

The following lexicon is used to describe the syntax of the Linker Script Language:

`A ::= B`	=	*A* is defined as *B*
`A ::= B C`	=	*A* is defined as *B* and *C*; *B* is followed by *C*
`A ::= B │ C`	=	*A* is defined as *B* or *C*
`<B>`$^{0\|1}$	=	zero or one occurrence of *B*
`<B>`$^{>=0}$	=	zero of more occurrences of *B*
`<B>`$^{>=1}$	=	one of more occurrences of *B*
`IDENTIFIER`	=	a character sequence starting with 'a'-'z', 'A'-'Z' or '_'. Following characters may also be digits and dots '.'
`STRING`	=	sequence of characters not starting with \n, \r or \t
`DQSTRING`	=	`"` `STRING` `"` (double quoted string)
`OCT_NUM`	=	octal number, starting with a zero (`06`, `045`)
`DEC_NUM`	=	decimal number, not starting with a zero (`14`, `1024`)
`HEX_NUM`	=	hexadecimal number, starting with '0x' (`0x0023`, `0xFF00`)

`OCT_NUM`, `DEC_NUM` and `HEX_NUM` can be followed by a **k** (kilo), **M** (mega), or **G** (giga).

Characters in **bold** are characters that occur literally. Words in *italics* are higher order terms that are defined in the same or in one of the other sections.

To write comments in LSL file, you can use the C style '`/* */`' or C++ style '`//`'.

16.2.3. Identifiers and Tags

```
arch_name          ::= IDENTIFIER
bus_name           ::= IDENTIFIER
core_name          ::= IDENTIFIER
derivative_name    ::= IDENTIFIER
file_name          ::= DQSTRING
group_name         ::= IDENTIFIER
heap_name          ::= section_name
map_name           ::= IDENTIFIER
mem_name           ::= IDENTIFIER
proc_name          ::= IDENTIFIER
section_name       ::= DQSTRING
space_name         ::= IDENTIFIER
stack_name         ::= section_name
symbol_name        ::= DQSTRING
```

```
tag_attr            ::= (tag<,tag>>=0)
tag                 ::= tag = DQSTRING
```

A tag is an arbitrary text that can be added to a statement.

16.2.4. Expressions

The expressions and operators in this section work the same as in ISO C.

```
number              ::= OCT_NUM
                      | DEC_NUM
                      | HEX_NUM

expr                ::= number
                      | symbol_name
                      | unary_op expr
                      | expr binary_op expr
                      | expr ? expr : expr
                      | ( expr )
                      | function_call

unary_op            ::= !    // logical NOT
                      | ~    // bitwise complement
                      | -    // negative value

binary_op           ::= ^    // exclusive OR
                      | *    // multiplication
                      | /    // division
                      | %    // modulus
                      | +    // addition
                      | -    // subtraction
                      | >>   // right shift
                      | <<   // left shift
                      | ==   // equal to
                      | !=   // not equal to
                      | >    // greater than
                      | <    // less than
                      | >=   // greater than or equal to
                      | <=   // less than or equal to
                      | &    // bitwise AND
                      | |    // bitwise OR
                      | &&   // logical AND
                      | ||   // logical OR
```

16.2.5. Built-in Functions

```
function_call       ::= absolute ( expr )
                      | addressof ( addr_id )
                      | checksum ( checksum_algo , expr , expr )
                      | exists ( section_name )
                      | max ( expr , expr )
```

```
                        |  min ( expr , expr )
                        |  sizeof ( size_id )

addr_id             ::=  sect : section_name
                        |  group : group_name

checksum_algo       ::=  crc32w

size_id             ::=  sect : section_name
                        |  group : group_name
                        |  mem : mem_name
```

- Every space, bus, memory, section or group you refer to, must be defined in the LSL file.

- The `addressof()` and `sizeof()` functions with the **group** or **sect** argument can only be used in the right hand side of an assignment. The `sizeof()` function with the **mem** argument can be used anywhere in section layouts.

- The `checksum()` function can only be used in a **struct** statement.

You can use the following built-in functions in expressions. All functions return a numerical value. This value is a 64-bit signed integer.

absolute()

```
int absolute( expr )
```

Converts the value of *expr* to a positive integer.

```
absolute( "labelA"-"labelB" )
```

addressof()

```
int addressof( addr_id )
```

Returns the address of *addr_id*, which is a named section or group. To get the offset of the section with the name `asect`:

```
addressof( sect: "asect")
```

> This function only works in assignments and **struct** statements.

checksum()

```
int checksum( checksum_algo, expr, expr )
```

Returns the computed checksum over a contiguous address range. The first argument specifies how the checksum must be computed (see below), the second argument is an expression that represents the start address of the range, while the third argument represents the end address (exclusive). The value of the end address expression must be strictly larger than the value of the start address (i.e. the size of the checksum address range must be at least one MAU). Each address in the range must point to a valid

memory location. Memory locations in the address range that are not occupied by a section are filled with zeros.

The only checksum algorithm (*checksum_algo*) currently supported is **crc32w**. This algorithm computes the checksum using a Cyclic Redundancy Check with the "CRC-32" polynomial 0xEDB88320. The input range is processed per 4-byte word. Those 4 bytes are passed to the checksum algorithm in reverse order if the target architecture is little-endian. For big-endian targets, this checksum algorithm is equal to a regular byte-wise CRC-32 implementation. Both the start address and end address values must be aligned on 4 MAUs. The behavior of this checksum algorithm is undefined when used in an address space that has a MAU size not equal to 8.

```
checksum( crc32w,
    __BMHD_ALIGN( addressof( mem:foo ), -4 ),
    __BMHD_ALIGN( addressof( mem:foo ) + sizeof( mem:foo ), 4 ) )
```

This function only works in **struct** statements.

exists()

```
int exists( section_name )
```

The function returns 1 if the section *section_name* exists in one or more object file, 0 otherwise. If the section is not present in input object files, but generated from LSL, the result of this function is undefined.

To check whether the section `mysection` exists in one of the object files that is specified to the linker:

```
exists( "mysection" )
```

max()

```
int max( expr, expr )
```

Returns the value of the expression that has the largest value. To get the highest value of two symbols:

```
max( "sym1" , "sym2")
```

min()

```
int min( expr, expr )
```

Returns the value of the expression hat has the smallest value. To get the lowest value of two symbols:

```
min( "sym1" , "sym2")
```

sizeof()

```
int sizeof( size_id )
```

Returns the size of the object (group, section or memory) the identifier refers to. To get the size of the section "asection":

```
sizeof( sect: "asection" )
```

> The **group** and **sect** arguments only works in assignments and **struct** statements. The **mem** argument can be used anywhere in section layouts.

16.2.6. LSL Definitions in the Linker Script File

description   ::= *<definition>*$^{>=1}$

definition   ::= *architecture_definition*
                     | *derivative_definition*
                     | *board_spec*
                     | *section_definition*
                     | *section_setup*

- At least one *architecture_definition* must be present in the LSL file.

16.2.7. Memory and Bus Definitions

mem_def   ::= **memory** *mem_name* *<tag_attr>*$^{0|1}$ **{**   *<mem_descr* **;***>*$^{>=0}$ **}**

- A *mem_def* defines a memory with the *mem_name* as a unique name.

mem_descr   ::= **type =** *<***reserved***>*$^{0|1}$ *mem_type*
                     | **mau =** *expr*
                     | **size =** *expr*
                     | **speed =** *number*
                     | **priority =** *number*
                     | **exec_priority =** *number*
                     | **fill <=** *fill_values*$^{0|1}$
                     | **write_unit =** *expr*
                     | *mapping*

- A *mem_def* contains exactly one **type** statement.

- A *mem_def* contains exactly one **mau** statement (non-zero size).

- A *mem_def* contains exactly one **size** statement.

- A *mem_def* contains zero or one **priority** (or **speed**) statement (if absent, the default value is 1).

- A *mem_def* contains zero or one **exec_priority** statement.

- A *mem_def* contains zero or one **fill** statement.

- A *mem_def* contains zero or one **write_unit** statement.

- A *mem_def* contains at least one *mapping*

mem_type   ::= **rom**   // attrs = rx
                     | **ram**   // attrs = rw

```
                      | nvram        // attrs = rwx
                      | blockram
```

fill_values   ::= *expr*
                     | **[** *expr* **<,** *expr***>**$^{>=0}$ **]**

bus_def   ::= **bus** *bus_name* **{** **<***bus_descr* **;>**$^{>=0}$ **}**

- A *bus_def* statement defines a bus with the given *bus_name* as a unique name within a core architecture.

bus_descr   ::= **mau =** *expr*
                     | **width =** *expr* // bus width, nr
                     | // of data bits
                     | *mapping*   // legal destination
                     | // 'bus' only

- The **mau** and **width** statements appear exactly once in a *bus_descr*. The default value for **width** is the **mau** size.

- The bus width must be an integer times the bus MAU size.

- The MAU size must be non-zero.

- A bus can only have a *mapping* on a destination bus (through **dest = bus:**).

mapping   ::= **map** **<***map_name***>**$^{0|1}$ **(** *map_descr* **<,** *map_descr***>**$^{>=0}$ **)**

map_descr   ::= **dest =** *destination*
                     | **dest_dbits =** *range*
                     | **dest_offset =** *expr*
                     | **size =** *expr*
                     | **src_dbits =** *range*
                     | **src_offset =** *expr*
                     | **reserved**
                     | **priority =** *number*
                     | **exec_priority =** *number*
                     | *tag*

- A *map_descr* requires at least the **size** and **dest** statements.

- A *map_descr* contains zero or one **priority** statement (if absent, the default value is 0).

- A *map_descr* contains zero or one **exec_priority** statement.

- Each *map_descr* can occur only once.

- You can define multiple mappings from a single source.

- Overlap between source ranges or destination ranges is not allowed.

- If the **src_dbits** or **dest_dbits** statement is not present, its value defaults to the **width** value if the source/destination is a bus, and to the **mau** size otherwise.

- The **reserved** statement is allowed only in mappings defined for a memory.

```
destination        ::= space : space_name
                     | bus : <proc_name |
                             core_name :>0|1 bus_name
```

- A `space_name` refers to a defined address space.

- A `proc_name` refers to a defined processor.

- A `core_name` refers to a defined core.

- A `bus_name` refers to a defined bus.

- The following mappings are allowed (source to destination)

  - space => space

  - space => bus

  - bus => bus

  - memory => bus

```
range              ::= expr .. expr
```

- With address ranges, the end address is not part of the range.

16.2.8. Architecture Definition

```
architecture_definition
                   ::= architecture arch_name
                       <( parameter_list )>0|1
                       <extends arch_name
                               <( argument_list )>0|1 >0|1
                       { <arch_spec>>=0 }
```

- An `architecture_definition` defines a core architecture with the given `arch_name` as a unique name.

- At least one `space_def` and at least one `bus_def` have to be present in an `architecture_definition`.

- An `architecture_definition` that uses the **extends** construct defines an architecture that inherits all elements of the architecture defined by the second `arch_name`. The parent architecture must be defined in the LSL file as well.

```
parameter_list     ::= parameter <, parameter>>=0

parameter          ::= IDENTIFIER <= expr>0|1

argument_list      ::= expr <, expr>>=0
```

```
arch_spec            ::= bus_def
                       | space_def
                       | endianness_def

space_def            ::= space space_name <tag_attr>^{0|1} { <space_descr;>^{>=0} }
```

- A *space_def* defines an address space with the given *space_name* as a unique name within an architecture.

```
space_descr          ::= space_property ;
                       | section_definition  //no space ref
                       | vector_table_statement
                       | reserved_range

space_property       ::= id = number // as used in object
                       | mau = expr
                       | align = expr
                       | page_size = expr <[ range ] <| [ range ]>^{>=0}>^{0|1}
                       | page
                       | direction = direction
                       | stack_def
                       | heap_def
                       | copy_table_def
                       | start_address
                       | mapping
```

- A *space_def* contains exactly one **id** and one **mau** statement.

- A *space_def* contains at most one **align** statement.

- A *space_def* contains at most one **page_size** statement.

- A *space_def* contains at least one *mapping*.

```
stack_def            ::= stack stack_name ( stack_heap_descr
                              <, stack_heap_descr >^{>=0} )
```

- A *stack_def* defines a stack with the *stack_name* as a unique name.

```
heap_def             ::= heap heap_name ( stack_heap_descr
                              <, stack_heap_descr >^{>=0} )
```

- A *heap_def* defines a heap with the *heap_name* as a unique name.

```
stack_heap_descr     ::= min_size = expr
                       | grows = direction
                       | align = expr
                       | fixed
                       | id = expr
                       | tag
```

- The **min_size** statement must be present.

- You can specify at most one **align** statement and one **grows** statement.

- Each stack definition has its own unique **id**, the number specified corresponds to the index in the `.CALLS` directive as generated by the compiler.

```
direction          ::= low_to_high
                     | high_to_low
```

- If you do not specify the **grows** statement, the stack and heap grow **low-to-high**.

```
copy_table_def     ::= copytable <( copy_table_descr
                            <, copy_table_descr >>=0 )>0|1
```

- A *space_def* contains at most one **copytable** statement.

- Exactly one copy table must be defined in one of the spaces.

```
copy_table_descr   ::= align = expr
                     | copy_unit = expr
                     | dest <space_name>0|1 = space_name
                     | page
                     | table { <subtable_descr; >>=0 }
                     | tag
```

```
subtable_descr     ::= symbol = symbol_name
                     | space = subtable_space_ref
```

```
subtable_space_ref
                   ::= <processor_name>0|1 : <core_name>0|1 : space_name
```

- The **copy_unit** is defined by the size in MAUs in which the startup code moves data.

- The **dest** statement is only required when the startup code initializes memory used by another processor that has no access to ROM.

- A *space_name* refers to a defined address space.

```
start_addr         ::= start_address ( start_addr_descr
                            <, start_addr_descr>>=0 )
```

```
start_addr_descr   ::= run_addr = expr
                     | symbol = symbol_name
```

- A *symbol_name* refers to the section that contains the startup code.

```
vector_table_statement
                   ::= vector_table section_name
                       ( vecttab_spec <, vecttab_spec>>=0 )
                       { <vector_def>>=0 }
```

```
vecttab_spec       ::= vector_size = expr
                     | size = expr
                     | id_symbol_prefix = symbol_name
```

```
                           |  run_addr = addr_absolute
                           |  template = section_name
                           |  template_symbol = symbol_name
                           |  vector_prefix = section_name
                           |  fill = vector_value
                           |  no_inline
                           |  copy
                           |  tag
```

vector_def   ::= **vector (** *vector_spec* **<,** *vector_spec*>$^{>=0}$ **);**

vector_spec   ::= **id =** *vector_id_spec*
                           | **fill =** *vector_value*
                           | **optional**
                           | *tag*

vector_id_spec   ::= *number*
                           | **[** *range* **] <, [** *range* **]**>$^{>=0}$

vector_value   ::= *symbol_name*
                           | **[** *number* **<,** *number*>$^{>=0}$ **]**
                           | **loop <[** *expr* **]**>$^{0|1}$

reserved_range   ::= **reserved** <*tag_attr*>$^{0|1}$ *expr* **..** *expr* **;**

- The end address is not part of the range.

endianness_def   ::= **endianness {** <*endianness_type*;>$^{>=1}$ **}**

endianness_type   ::= **big**
                           | **little**

16.2.9. Derivative Definition

derivative_definition
                      ::= **derivative** *derivative_name*
                           <**(** *parameter_list* **)**>$^{0|1}$
                           <**extends** *derivative_name*
                                   <**(** *argument_list* **)**>$^{0|1}$ >$^{0|1}$
                           { <*derivative_spec*>$^{>=0}$ **}**

- A *derivative_definition* defines a derivative with the given *derivative_name* as a unique name.

derivative_spec   ::= *core_def*
                           | *bus_def*
                           | *mem_def*
                           | *section_definition* // no processor name
                           | *section_setup*

core_def   ::= **core** *core_name* **{** <*core_descr* **;**>$^{>=0}$ **}**

- A `core_def` defines a core with the given `core_name` as a unique name.

- At least one `core_def` must be present in a `derivative_definition`.

```
core_descr          ::= architecture = arch_name
                        <( argument_list )>^{0|1}
                      | copytable_space <core_name :>^{0|1} space_name
                      | endianness = ( endianness_type
                                <, endianness_type>^{>=0} )
                      | import core_name
                      | space_id_offset = number
```

- An `arch_name` refers to a defined core architecture.

- Exactly one **architecture** statement must be present in a `core_def`.

- Exactly one **copytable_space** statement must be present in a `core_def`, or in exactly one space in that core, a **copytable** statement must be present.

16.2.10. Processor Definition and Board Specification

```
board_spec          ::= proc_def
                      | bus_def
                      | mem_def

proc_def            ::= processor proc_name
                        { proc_descr ; }

proc_descr          ::= derivative = derivative_name
                        <( argument_list )>^{0|1}
```

- A `proc_def` defines a processor with the `proc_name` as a unique name.

- If you do not explicitly define a processor for a derivative in an LSL file, the linker defines a processor with the same name as that derivative.

- A `derivative_name` refers to a defined derivative.

- A `proc_def` contains exactly one **derivative** statement.

16.2.11. Section Setup

```
section_setup       ::= section_setup space_ref <tag_attr>^{0|1}
                        { <section_setup_item>^{>=0} }

section_setup_item
                    ::= vector_table_statement
                      | reserved_range
                      | stack_def ;
                      | heap_def ;
                      | copy_table_def ;
                      | start_address ;
```

```
                      | reference_space_restriction ;
                      | modify linktime_modification
                      | section_reference_restriction

reference_space_restriction
                  ::= prohibit_references_to subtable_space_ref
                      <,subtable_space_ref><sup></sup>
```

reference_space_restriction

::= **prohibit_references_to** *subtable_space_ref*
    <,*subtable_space_ref*>$^{>=0}$

linktime_modification

::= **input (** *input_modifier* <,*input_modifier*>$^{>=0}$ **)**
    { <*select_section_statement* **;** >$^{>=0}$ **}**

input_modifier   ::= **space =** *subtable_space_ref*
                      | **attributes = <** <**+**|**-**> *attribute*>$^{>0}$
                      | **copy**

- An *input_modifier* contains at most one **space** statement.

- An *input_modifier* contains at most one **attributes** statement.

section_reference_restriction

::= **section_reference_restriction**
    { <*section_reference_restriction_item* **;** >$^{>=0}$ **}**

section_reference_restriction_item

::= **safety_class =** *number*
    | **target_safety_class =** *number*
    | **attributes =** <*attribute*>$^{>0}$

- Each *section_reference_restriction_item* can occur at most once in a
  section_reference_restriction statement.

16.2.12. Section Layout Definition

section_definition ::= **section_layout** <*space_ref*>$^{0|1}$
                      <**(** *space_layout_properties* **)**>$^{0|1}$
                      { <*section_statement*>$^{>=0}$ **}**

- A section definition inside a space definition does not have a *space_ref*.

- All global section definitions have a *space_ref*.

space_ref   ::= <*proc_name*>$^{0|1}$ **:** <*core_name*>$^{0|1}$
                      : *space_name* <**|** *space_name*>$^{>=0}$

- If more than one processor is present, the *proc_name* must be given for a global section layout.

- If the section layout refers to a processor that has more than one core, the *core_name* must be given
  in the *space_ref*.

- A *proc_name* refers to a defined processor.

- A `core_name` refers to a defined core.

- A `space_name` refers to a defined address space.

`space_layout_properties`
$$::= space\_layout\_property <, space\_layout\_property >^{>=0}$$

`space_layout_property`
$$::= locate\_direction$$
$$| \; tag$$

`locate_direction` ::= **direction =** *direction*

`direction`   ::= **low_to_high**
                       | **high_to_low**

- A section layout contains at most one **direction** statement.

- If you do not specify the **direction** statement, the locate direction of the section layout is **low-to-high**.

`section_statement`
$$::= simple\_section\_statement \; ;$$
$$| \; aggregate\_section\_statement$$

`simple_section_statement`
$$::= assignment$$
$$| \; select\_section\_statement$$
$$| \; special\_section\_statement$$
$$| \; memcopy\_statement$$

`assignment`   ::= *symbol_name assign_op expr*

`assign_op`   ::= **=**
                       | **:=**

`select_section_statement`
$$::= \textbf{select} <\textbf{ref\_tree}>^{0|1} <section\_name>^{0|1}$$
$$<section\_selections>^{0|1}$$

- Either a `section_name` or at least one `section_selection` must be defined.

`section_selections`
$$::= \textbf{(} section\_selection$$
$$<, section\_selection>^{>=0} \textbf{)}$$

`section_selection`
$$::= \textbf{attributes =} < <+|-> attribute>^{>0}$$
$$| \; tag$$

- **+**_attribute_ means: select all sections that have this attribute.

- **-**_attribute_ means: select all sections that do not have this attribute.

```
special_section_statement
                 ::= heap heap_name <stack_heap_mods>^{0|1}
                   | stack stack_name <stack_heap_mods>^{0|1}
                   | copytable
                   | reserved section_name <reserved_specs>^{0|1}
```

- Special sections cannot be selected in load-time groups.

```
stack_heap_mods    ::= ( stack_heap_mod <, stack_heap_mod>^{>=0} )

stack_heap_mod     ::= size = expr
                     | tag

reserved_specs     ::= ( reserved_spec <, reserved_spec>^{>=0} )

reserved_spec      ::= attributes
                     | fill_spec
                     | size = expr
                     | alloc_allowed = absolute | ranged
```

- If a **reserved** section has attributes **r**, **rw**, **x**, **rx** or **rwx**, and no fill pattern is defined, the section is filled with zeros. If no attributes are set, the section is created as a scratch section (attributes **ws**, no image).

```
memcopy_statement
                 ::= memcopy section_name
                     ( memcopy_spec <, memcopy_spec>^{0|1} )

memcopy_spec       ::= memory = memory_reference
                     | fill_spec
```

- A **memcopy** statement must contain exactly one **memory** statement.

- A **memcopy** statement can contain at most one *fill_spec*.

```
fill_spec          ::= fill = fill_values

fill_values        ::= expr
                     | [ expr <, expr>^{>=0} ]

aggregate_section_statement
                 ::= { <section_statement>^{>=0} }
                   | group_descr
                   | if_statement
                   | section_creation_statement
                   | struct_statement

group_descr        ::= group <group_name>^{0|1} <( group_specs )>^{0|1}
                             section_statement
```

- For every group with a name, the linker defines a label.

- No two groups for address spaces of a core can have the same *group_name*.

```
group_specs          ::= group_spec <, group_spec >>=0

group_spec           ::= group_alignment
                       | attributes
                       | copy
                       | nocopy
                       | group_load_address
                       | fill <= fill_values>0|1
                       | group_page
                       | group_run_address
                       | group_type
                       | allow_cross_references
                       | priority = number
                       | group_safety_class
                       | tag
```

- The **allow-cross-references** property is only allowed for *overlay* groups.

- Sub groups inherit all properties from a parent group.

```
group_alignment      ::= align = expr

attributes           ::= attributes = <attribute>>=1

attribute            ::= r    // readable sections
                       | w    // writable sections
                       | x    // executable code sections
                       | i    // initialized sections
                       | s    // scratch sections
                       | b    // blanked (cleared) sections
                       | p    // protected sections

group_load_address
                     ::= load_addr <= load_or_run_addr>0|1

group_page           ::= page <= expr>0|1
                       | page_size = expr <[ range ] <| [ range ]>>=0>0|1

group_run_address    ::= run_addr <= load_or_run_addr>0|1

group_type           ::= clustered
                       | contiguous
                       | ordered
                       | overlay
```

- For *non-contiguous* groups, you can only specify `group_alignment` and `attributes`.

- The **overlay** keyword also sets the **contiguous** property.

- The **clustered** property cannot be set together with **contiguous** or **ordered** on a single group.

group_safety_class
                    ::= **safety_class =** *number*

load_or_run_addr   ::= *addr_absolute*
                    | *addr_range* <| *addr_range*>$^{>=0}$

addr_absolute   ::= *expr*
                    | *memory_reference* **[** *expr* **]**

- An absolute address can only be set on *ordered* groups.

addr_range   ::= **[** *expr* **..** *expr* **]**
                    | *memory_reference*
                    | *memory_reference* **[** *expr* **..** *expr* **]**

- The parent of a group with an *addr_range* or **page** restriction cannot be **ordered**, **contiguous** or **clustered**.

- The end address is not part of the range.

memory_reference   ::= **mem :** <*proc_name* **:**>$^{0|1}$ *mem_name* </ *map_name*>$^{0|1}$

- A *proc_name* refers to a defined processor.

- A *mem_name* refers to a defined memory.

- A *map_name* refers to a defined memory mapping.

if_statement   ::= **if (** *expr* **)** *section_statement*
                    <**else** *section_statement*>$^{0|1}$

section_creation_statement
                    ::= **section** *section_name* **(** *section_specs* **)**
                    { <*section_statement2*>$^{>=0}$ **}**

section_specs   ::= *section_spec* <**,** *section_spec* >$^{>=0}$

section_spec   ::= *attributes*
                    | *fill_spec*
                    | **size =** *expr*
                    | **blocksize =** *expr*
                    | **overflow =** *section_name*
                    | *tag*

section_statement2
                    ::= *select_section_statement* **;**
                    | *group_descr2*
                    | **{** <*section_statement2*>$^{>=0}$ **}**

group_descr2   ::= **group** <*group_name*>$^{0|1}$
                        (*group_specs2* **)**
                         section_statement2

```
group_specs2        ::= group_spec2 <, group_spec2 >>=0

group_spec2         ::= group_alignment
                      | attributes
                      | load_addr
                      | tag

struct_statement
                    ::= struct { <struct_item>>=0 }

struct_item         ::= expr : number ;
```

16.3. Expression Evaluation

Only *constant* expressions are allowed, including sizes, but not addresses, of sections in object files.

All expressions are evaluated with 64-bit precision integer arithmetic. The result of an expression can be absolute or relocatable. A symbol you assign is created as an absolute symbol. Symbol references are only allowed in symbol assignments and **struct** statements.

16.4. Semantics of the Architecture Definition

Keywords in the architecture definition

```
architecture
   extends
endianness        big  little
bus
   mau
   width
   map
space
   id
   mau
   align
   page_size
   page
   direction      low_to_high  high_to_low
   stack
      min_size
      grows        low_to_high  high_to_low
      align
      fixed
      id
   heap
      min_size
      grows        low_to_high  high_to_low
      align
```

```
      fixed
      id
   copytable
      align
      copy_unit
      dest
      page
      table          space   symbol
   vector_table
      vector_size
      size
      id_symbol_prefix
      run_addr
      template
      template_symbol
      vector_prefix
      fill
      no_inline
      copy
      vector
         id
         fill          loop
         optional
   reserved
   start_address
      run_addr
      symbol
   map

   map
      dest           bus   space
      dest_dbits
      dest_offset
      size
      src_dbits
      src_offset
      priority
      exec_priority
```

16.4.1. Defining an Architecture

With the keyword **architecture** you define an architecture and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

```
architecture name
{
    definitions
}
```

If you are defining multiple core architectures that show great resemblance, you can define the common features in a parent core architecture and extend this with a child core architecture that contains specific features. The child inherits all features of the parent. With the keyword extends you create a child core architecture:

```
architecture name_child_arch extends name_parent_arch
{
    definitions
}
```

A core architecture can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the architecture. You can use them in any expression within the core architecture. Parameters can have default values, which are used when the core architecture is instantiated with less arguments than there are parameters defined for it. When you extend a core architecture you can pass arguments to the parent architecture. Arguments are expressions that set the value of the parameters of the sub-architecture.

```
architecture name_child_arch (parm1,parm2=1)
             extends name_parent_arch (arguments)
{
    definitions
}
```

16.4.2. Defining Internal Buses

With the **bus** keyword you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions in an architecture definition or derivative definition define *internal* buses. Some internal buses are used to communicate with the components outside the core or processor. Such buses on a processor have physical pins reserved for the number of bits specified with the **width** statements.

- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the data bus. This field is required.

- The **width** field specifies the width (number of address lines) of the data bus. The default value is the MAU size.

- The **map** keyword specifies how this bus maps onto another bus (if so). Mappings are described in Section 16.4.4, *Mappings*.

```
bus bus_name
{
    mau = 8;
    width = 8;
    map ( map_description );
}
```

16.4.3. Defining Address Spaces

With the **space** keyword you define a logical address space. The space name is used to identify the address space and does not conflict with other identifiers.

- The `id` field defines how the addressing space is identified in object files. In general, each address space has a unique ID. The linker locates sections with a certain ID in the address space with the same ID. This field is required.

- The `mau` field specifies the MAU size (Minimum Addressable Unit) of the space. This field is required.

- The `align` value must be a power of two. The linker uses this value to compute the start addresses when sections are concatenated. An align value of *n* means that objects in the address space have to be aligned on *n* MAUs.

- The `page_size` field sets the page alignment and page size in MAUs for the address space. It must be a power of 2. The default value is 1. If one or more page ranges are supplied the supplied value only sets the page alignment. The ranges specify the available space in each page, as offsets to the page start, which is aligned at the page alignment.

  See also the `page` keyword in subsection Locating a group in Section 16.8.2, *Creating and Locating Groups of Sections*.

- With the optional `direction` field you can specify how all sections in this space should be located. This can be either from `low_to_high` addresses (this is the default) or from `high_to_low` addresses.

- The `map` keyword specifies how this address space maps onto an internal bus or onto another address space. Mappings are described in Section 16.4.4, *Mappings*.

Stacks and heaps

- The `stack` keyword defines a stack in the address space and assigns a name to it. The architecture definition must contain at least one stack definition. Each stack of a core architecture must have a unique name. See also the `stack` keyword in Section 16.8.3, *Creating or Modifying Special Sections*.

  The stack is described in terms of a minimum size (`min_size`) and the direction in which the stack grows (`grows`). This can be either from `low_to_high` addresses (stack grows upwards, this is the default) or from `high_to_low` addresses (stack grows downwards). The `min_size` is required.

  By default, the linker tries to maximize the size of the stacks and heaps. After locating all sections, the largest remaining gap in the space is used completely for the stacks and heaps. If you specify the keyword `fixed`, you can disable this so-called 'balloon behavior'. The size is also fixed if you used a stack or heap in the software layout definition in a restricted way. For example when you override a stack with another size or select a stack in an ordered group with other sections.

  The `id` keyword matches stack information generated by the compiler with a stack name specified in LSL. This value assigned to this keyword is strongly related to the compiler's output, so users are not supposed to change this configuration.

  Optionally you can specify an alignment for the stack with the argument `align`. This alignment must be equal or larger than the alignment that you specify for the address space itself.

- The `heap` keyword defines a heap in the address space and assigns a name to it. The definition of a heap is similar to the definition of a stack. See also the `heap` keyword in Section 16.8.3, *Creating or Modifying Special Sections*.

Stacks and heaps are only generated by the linker if the corresponding linker labels are referenced in the object files.

See Section 16.8, *Semantics of the Section Layout Definition*, for information on creating and placing stack sections.

Copy tables

- The **copytable** keyword defines a copy table in the address space. The content of the copy table is created by the linker and contains the start address and size of all sections that should be initialized by the startup code. You must define exactly one copy table in one of the address spaces (for a core).

  Optionally you can specify an alignment for the copy table with the argument **align**. This alignment must be equal or larger than the alignment that you specify for the address space itself. If smaller, the alignment for the address space is used.

  The **copy_unit** argument specifies the size in MAUs of information chunks that are copied. If you do not specify the copy unit, the MAU size of the address space itself is used.

  The **dest** argument specifies the destination address space that the code uses for the copy table. The linker uses this information to generate the correct addresses in the copy table. The memory into where the sections must be copied at run-time, must be accessible from this destination space.

  Sections generated for the copy table may get a page restriction with the address space's page size, by adding the **page** argument.

  One or more **table** arguments split off one or more sub-tables from a copy table. Each sub-table can be handled separately from the others and from the main table. This can be useful in a multi-core system, for example. All initialization entries generated from a section in an address space can be redirected to a sub-table by putting the address space name in a comma-separated list following **space =**. The initialization code that handles a sub-table needs a reference to it. This can be accomplished by specifying a symbol for the sub-table using **symbol =** *symbol_name***;**. Each sub-table including the main table (which is filled with all entries not redirected to a sub-table) ends with a terminator entry and they are all placed in the regular copy table section.

```
copytable
(
    align = 4,
    dest = linear,
    table
    {
        symbol = "_lc_ub_table_tc1";
        space = :tc1:linear, :tc1:abs24, :tc1:abs18, :tc1:csa;
    },
    table
    {
        symbol = "_lc_ub_table_tc2";
        space = :tc2:linear, :tc2:abs24, :tc2:abs18, :tc2:csa;
    }
);
```

Vector table

• The **vector_table** keyword defines a vector table with *n* vectors of size *m* (This is an internal LSL object similar to an LSL group.) The **run_addr** argument specifies the location of the first vector (id=0). This can be a simple address or an offset in memory (see the description of the run-time address in subsection Locating a group in Section 16.8.2, *Creating and Locating Groups of Sections*). A vector table defines symbols _lc_ub_foo and _lc_ue_foo pointing to start and end of the table.

```
vector_table "vtable" (vector_size=m, size=n, run_addr=x, ...)
```

See the following example of a vector table definition:

```
vector_table "vtable" (vector_size = 4, size = 256, run_addr=0,
                   template=".text.vector_template",
                   template_symbol="_lc_vector_target",
                   vector_prefix=".text.vector.",
                   id_symbol_prefix="foo",
                   no_inline,
                   /* default: empty, or */
                   fill="foo", /* or */
                   fill=[1,2,3,4], /* or */
                   fill=loop)
{
    vector (id=23, fill="main", optional);
    vector (id=12, fill=[0xab, 0x21, 0x32, 0x43]);
    vector (id=[1..11], fill=[0]);
    vector (id=[18..23], fill=loop);
}
```

The **template** argument defines the name of the section that holds the code to jump to a handler function from the vector table. This template section does not get located and is removed when the locate phase is completed. This argument is required.

The **template_symbol** argument is the symbol reference in the template section that must be replaced by the address of the handler function. This symbol name should start with the linker prefix for the symbol to be ignored in the link phase. This argument is required.

The **vector_prefix** argument defines the names of vector sections: the section for a vector with id *vector_id* is $(*vector_prefix*)$(*vector_id*). Vectors defined in C or assembly source files that should be included in the vector table must have the correct symbol name. The compiler uses the prefix that is defined in the default LSL file(s); if this attribute is changed, the vectors declared in C source files are not included in the vector table. When a vector supplied in an object file has exactly one relocation, the linker will assume it is a branch to a handler function, and can be removed when the handler is inlined in the vector table. Otherwise, no inlining is done. This argument is required.

With the optional **no_inline** argument the vectors handlers are not inlined in the vector table.

With the optional **copy** argument a ROM copy of the vector table is made and the vector table is copied to RAM at startup.

With the optional **id_symbol_prefix** argument you can set an internal string representing a symbol name prefix that may be found on symbols in vector handler code. When the linker detects such a

symbol in a handler, the symbol is assigned the vector number. If the symbol was already assigned a vector number, a warning is issued.

The **fill** argument sets the default contents of vectors. If nothing is specified for a vector, this setting is used. See below. When no default is provided, empty vectors may be used to locate large vector handlers and other sections. Only one **fill** argument is allowed.

The **vector** field defines the content of vector with the number specified by **id**. If a range is specified for **id** ([p..q,s..t]) all vectors in the ranges (inclusive) are defined the same way.

With **fill=***symbol_name*, the vector must jump to this symbol. If the section in which the symbol is defined fits in the vector table (size may be >*m*), locate the section at the location of the vector. Otherwise, insert code to jump to the symbol's value. A template interrupt handler section name + symbol name for the target code must be supplied in the LSL file.

fill=[*value(s)***]**, fills the vector with the specified MAU values.

With **fill=loop** the vector jumps to itself. With the optional **[***offset***]** you can specify an offset from the vector table entry.

When the keyword **optional** is set on a vector specification with a symbol value and the symbol is not found, no error is reported. A default fill value is used if the symbol was not found. With other values the attribute has no effect.

Reserved address ranges

- The **reserved** keyword specifies to reserve a part of an address space even if not all of the range is covered by memory. See also the **reserved** keyword in Section 16.8.3, *Creating or Modifying Special Sections*.

Start address

- The **start_address** keyword specifies the start address for the position where the C startup code is located. When a processor is reset, it initializes its program counter to a certain start address, sometimes called the reset vector. In the architecture definition, you must specify this start address in the correct address space in combination with the name of the label in the application code which must be located here.

  The **run_addr** argument specifies the start address (reset vector). If the core starts executing using an entry from a vector table, and directly jumps to the start label, you should omit this argument.

  The **symbol** argument specifies the name of the label in the application code that should be located at the specified start address. The **symbol** argument is required. The linker will resolve the start symbol and use its value after locating for the start address field in IEEE-695 files and Intel Hex files. If you also specified the **run_addr** argument, the start symbol (label) must point to a section. The linker locates this section such that the start symbol ends up on the start address.

```
space space_name
{
    id = 1;
    mau = 8;
    align = 8;
```

```
        page_size = 1;
        stack name (min_size = 1k, grows = low_to_high);
        reserved start_address .. end_address;
        start_address ( run_addr = 0x0000,
                        symbol = "start_label" )
        map ( map_description );
}
```

16.4.4. Mappings

You can use a mapping when you define a space, bus or memory. With the **map** field you specify how addresses from the source (space, bus or memory) are translated to addresses of a destination (space, bus). The following mappings are possible:

- space => space

- space => bus

- bus => bus

- memory => bus

With a mapping you specify a range of source addresses you want to map (specified by a source offset and a size), the destination to which you want to map them (a bus or another address space), and the offset address in the destination.

- The **dest** argument specifies the destination. This can be a **bus** or another address **space** (only for a space to space mapping). This argument is required.

- The **src_offset** argument specifies the offset of the source addresses. In combination with size, this specifies the range of address that are mapped. By default the source offset is 0x0000.

- The **size** argument specifies the number of addresses that are mapped. This argument is required.

- The **dest_offset** argument specifies the position in the destination to which the specified range of addresses is mapped. By default the destination offset is 0x0000.

If you are mapping a bus to another bus, the number of data lines of each bus may differ. In this case you have to specify a range of source data lines you want to map (**src_dbits =** *begin..end*) and the range of destination data lines you want to map them to (**dest_dbits =** *first..last*).

- The **src_dbits** argument specifies a range of data lines of the source bus. By default all data lines are mapped.

- The **dest_dbits** argument specifies a range of data lines of the destination bus. By default, all data lines from the source bus are mapped on the data lines of the destination bus (starting with line 0).

If you define a memory and the memory mapping must not be used by default when locating sections in address spaces, you can specify the **reserved** argument. This marks all address space areas that the mapping points to as reserved. If a section has an absolute or address range restriction, the reservation is lifted and the section may be located at these locations. This feature is only useful when more than

one mapping is available for a range of memory addresses, otherwise the **memory** keyword with the same name would be used.

For example:

```
memory xrom
{
    mau = 8;
    size = 1M;
    type = rom;
    map     cached (dest=bus:spe:fpi_bus, dest_offset=0x80000000,
                    size=1M);
    map not_cached (dest=bus:spe:fpi_bus, dest_offset=0xa0000000,
                    size=1M, reserved);
}
```

Mapping priority

If you define a memory you can set a locate priority on a mapping with the keywords **priority** and **exec_priority**. The values of these priorities are relative which means they add to the priority of memories. Whereas a priority set on the memory applies to all address space areas reachable through any mapping of the memory, a priority set on a mapping only applies to address space areas reachable through the mapping. The memory mapping with the highest priority is considered first when locating. To set only a priority for non-executable (data) sections, add a **priority** keyword with the desired value and an **exec_priority** set to zero. To set only a priority for executable (code) sections, simply set an **exec_priority** keyword to the desired value.

The default for a mapping **priority** is zero, while the default for **exec_priority** is the same as the specified **priority**. If you specify a value for **priority** in LSL it must be greater than zero. A value for **exec_priority** must be greater or equal to zero.

For more information about priority values see the description of the memory **priority** keyword.

```
memory dspram
{
    mau = 8;
    size = 112k;
    type = ram;
    map (dest=bus:tc0:fpi_bus, dest_offset=0xd0000000,
            size=112k, priority=8, exec_priority=0);
    map (dest=bus:sri, dest_offset=0x70000000,
            size=112k);
}
```

From space to space

If you map an address space to another address space (nesting), you can do this by mapping the subspace to the containing larger space. In this example a small space of 64 kB is mapped on a large space of 16 MB.

```
space small
{
   id = 2;
   mau = 4;
   map (src_offset = 0, dest_offset = 0,
        dest = space : large, size = 64k);
}
```

From space to bus

All spaces that are not mapped to another space must map to a bus in the architecture:

```
space large
{
   id = 1;
   mau = 4;
   map (src_offset = 0, dest_offset = 0,
        dest = bus:bus_name, size = 16M );
}
```

From bus to bus

The next example maps an external bus called e_bus to an internal bus called i_bus. This internal bus resides on a core called mycore. The source bus has 16 data lines whereas the destination bus has only 8 data lines. Therefore, the keywords **src_dbits** and **dest_dbits** specify which source data lines are mapped on which destination data lines.

```
architecture mycore
{
    bus i_bus
    {
       mau = 4;
    }

    space i_space
    {
       map (dest=bus:i_bus, size=256);
    }
}

bus e_bus
{
   mau = 16;
   width = 16;
   map (dest = bus:mycore:i_bus, src_dbits = 0..7, dest_dbits = 0..7 )
}
```

It is not possible to map an internal bus to an external bus.

16.5. Semantics of the Derivative Definition

Keywords in the derivative definition

```
derivative
   extends
core
   architecture
   import
   space_id_offset
   copytable_space
bus
   mau
   width
   map
memory
   type              reserved rom  ram  nvram  blockram
   mau
   size
   speed
   priority
   exec_priority
   fill
   write_unit
   map
section_layout
section_setup

   map
      dest          bus  space
      dest_dbits
      dest_offset
      size
      src_dbits
      src_offset
      priority
      exec_priority
      reserved
```

16.5.1. Defining a Derivative

With the keyword **derivative** you define a derivative and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

```
derivative name
{
    definitions
}
```

If you are defining multiple derivatives that show great resemblance, you can define the common features in a parent derivative and extend this with a child derivative that contains specific features. The child inherits all features of the parent (cores and memories). With the keyword **extends** you create a child derivative:

```
derivative name_child_deriv extends name_parent_deriv
{
    definitions
}
```

As with a core architecture, a derivative can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the derivative. You can use them in any expression within the derivative definition.

```
derivative name_child_deriv (parm1,parm2=1)
          extends name_parent_deriv (arguments)
{
    definitions
}
```

16.5.2. Instantiating Core Architectures

With the keyword **core** you instantiate a core architecture in a derivative.

- With the keyword **architecture** you tell the linker that the given core has a certain architecture. The architecture name refers to an existing architecture definition in the same LSL file.

  For example, if you have two cores (called mycore_1 and mycore_2) that have the same architecture (called mycorearch), you must instantiate both cores as follows:

  ```
  core mycore_1
  {
      architecture = mycorearch;
  }

  core mycore_2
  {
      architecture = mycorearch;
  }
  ```

  If the architecture definition has parameters you must specify the arguments that correspond with the parameters. For example mycorearch1 expects two parameters which are used in the architecture definition:

  ```
  core mycore
  {
      architecture = mycorearch1 (1,2);
  }
  ```

- With the keyword **import** you can combine multiple cores with the same architecture into a single link task. The imported cores share a single symbol namespace.

- The address spaces in each imported core must have a unique ID in the link task. With the keyword **space_id_offset** you specify for each imported core that the space IDs of the imported core start at a specific offset.

- With the keyword **copytable_space** you can specify that writable sections for a core must be initialized by using the copy table of a different core.

```
core mycore_1
{
    architecture = mycorearch;
    space_id_offset = 100; // add 100 to all space IDs in
                           // the architecture definition
    copytable_space = mycore:myspace; // use copytable from core mycore
}
core mycore_2
{
    architecture = mycorearch;
    space_id_offset = 200; // add 200 to all space IDs in
                           // the architecture definition
    copytable_space = mycore:myspace; // use copytable from core mycore
}

core mycore
{
    architecture = mycorearch;
    import mycore_1; // add all address spaces of mycore_1 for linking
    import mycore_2; // add all address spaces of mycore_2 for linking
}
```

16.5.3. Defining Internal Memory and Buses

With the keyword **memory** you define physical memory that is present on the target board. The memory name is used to identify the memory and does not conflict with other identifiers. It is common to define internal memory (on-chip) in the derivative definition. External memory (off-chip memory) is usually defined in the board specification (See Section 16.6.3, *Defining External Memory and Buses*).

- The **type** field specifies a memory type:

  - **rom**: read-only memory - it can only be written at load-time

  - **ram**: random access volatile writable memory - writing at run-time is possible while writing at load-time has no use since the data is not retained after a power-down

  - **nvram**: non volatile ram - writing is possible both at load-time and run-time

  - **blockram**: writing is possible both at load-time and run-time. Changes are applied in RAM, so after a full device reset the data in a blockram reverts to the original state.

The optional **reserved** qualifier before the memory type, tells the linker not to locate any section in the memory by default. You can locate sections in such memories using an absolute address or range restriction (see subsection Locating a group in Section 16.8.2, *Creating and Locating Groups of Sections*).

- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the memory. This field is required.

- The **size** field specifies the size in MAU of the memory. This field is required.

- The **priority** field specifies a locate priority for a memory. The **speed** field has the same meaning but is considered deprecated. By default, a memory has its priority set to 1. The memories with the highest priority are considered first when trying to locate a rule. Subsequently, the next highest priority memories are added if the rule was not located successfully, and so on until the lowest priority that is available is reached or the rule is located. The lowest priority value is zero. Sections with an **ordered** and/or **contiguous** restriction are not affected by the locate priority. If such sections also have a **page** restriction, the locate priority is still used to select a page.

- If an **exec_priority** is specified for a memory, the regular priority (either specified or its default value) does not apply to locate rules with only executable sections. Instead, the supplied value applies for such rules. Additionally, the **exec_priority** value is used for any executable unrestricted sections, even if they appear in an unrestricted rule together with non-executable sections.

- The **map** field specifies how this memory maps onto an (internal) bus. The mapping can have a name. Mappings are described in Section 16.4.4, *Mappings*.

- The optional **write_unit** field specifies the minimum write unit (MWU). This is the minimum number of MAUs required in a write action. This is useful to initialize memories that can only be written in units of two or more MAUs. If **write_unit** is not defined the minimum write unit is 0.

- The optional **fill** field contains a bit pattern that the linker writes to all memory addresses that remain unoccupied during the locate process. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU.

```
memory mem_name
{
    type = rom;
    mau = 8;
    write_unit = 4;
    fill = 0xaa;
    size = 64k;
    priority = 2;
    map map_name ( map_description );
}
```

With the **bus** keyword you define a bus in a derivative definition. Buses are described in Section 16.4.2, *Defining Internal Buses*.

16.6. Semantics of the Board Specification

Keywords in the board specification

```
processor
   derivative
bus
   mau
   width
   map
memory
   type              reserved  rom  ram  nvram  blockram
   mau
   size
   speed
   priority
   exec_priority
   fill
   write_unit
   map

   map
      dest            bus   space
      dest_dbits
      dest_offset
      size
      src_dbits
      src_offset
      priority
      exec_priority
      reserved
```

16.6.1. Defining a Processor

If you have a target board with multiple processors that have the same derivative, you need to instantiate each individual processor in a processor definition. This information tells the linker which processor has which derivative and enables the linker to distinguish between the present processors.

If you use processors that all have a unique derivative, you may omit the processor definitions. In this case the linker assumes that for each derivative definition in the LSL file there is one processor. The linker uses the derivative name also for the processor.

With the keyword **processor** you define a processor. You can freely choose the processor name. The name is used to refer to it at other places in the LSL file:

processor *proc_name*
{

```
    processor definition
}
```

16.6.2. Instantiating Derivatives

With the keyword **derivative** you tell the linker that the given processor has a certain derivative. The derivative name refers to an existing derivative definition in the same LSL file.

For example, if you have two processors on your target board (called myproc_1 and myproc_2) that have the same derivative (called myderiv), you must instantiate both processors as follows:

```
processor myproc_1
{
    derivative = myderiv;
}

processor myproc_2
{
    derivative = myderiv;
}
```

If the derivative definition has parameters you must specify the arguments that correspond with the parameters. For example myderiv1 expects two parameters which are used in the derivative definition:

```
processor myproc
{
    derivative = myderiv1 (2,4);
}
```

16.6.3. Defining External Memory and Buses

It is common to define external memory (off-chip) and external buses at the global scope (outside any enclosing definition). Internal memory (on-chip memory) is usually defined in the scope of a derivative definition.

With the keyword **memory** you define physical memory that is present on the target board. The memory name is used to identify the memory and does not conflict with other identifiers. If you define memory parts in the LSL file, only the memory defined in these parts is used for placing sections.

If no external memory is defined in the LSL file and if the linker option to allocate memory on demand is set then the linker will assume that all virtual addresses are mapped on physical memory. You can override this behavior by specifying one or more memory definitions.

```
memory mem_name
{
    type = rom;
    mau = 8;
    write_unit = 4;
    fill = 0xaa;
    size = 64k;
    priority = 2;
```

```
    map map_name ( map_description );
}
```

For a description of the keywords, see Section 16.5.3, *Defining Internal Memory and Buses*.

With the keyword **bus** you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions at the global scope (outside any definition) define external buses. These are buses that are present on the target board.

```
bus bus_name
{
    mau = 8;
    width = 8;
    map ( map_description );
}
```

For a description of the keywords, see Section 16.4.2, *Defining Internal Buses*.

You can connect off-chip memory to any derivative: you need to map the off-chip memory to a bus and map that bus on the internal bus of the derivative you want to connect it to.

16.7. Semantics of the Section Setup Definition

Keywords in the section setup definition

```
section_setup
   stack
      min_size
      grows          low_to_high  high_to_low
      align
      fixed
      id
   heap
      min_size
      grows          low_to_high  high_to_low
      align
      fixed
      id
   copytable
      align
      copy_unit
      dest
      page
      table          space   symbol
   vector_table
      vector_size
      size
      id_symbol_prefix
```

```
    run_addr
    template
    template_symbol
    vector_prefix
    fill
    no_inline
    copy
    vector
        id
        fill        loop
        optional
reserved
start_address
    run_addr
    symbol
prohibit_references_to
modify input
    space
    attributes
    copy
section_reference_restriction
    safety_class
    target_safety_class
    attributes
```

16.7.1. Setting up a Section

With the keyword **section_setup** you can define stacks, heaps, copy tables, vector tables, start address and/or reserved address ranges outside their address space definition. In addition you can configure space reference restrictions, input section modifications and section reference restrictions.

```
section_setup ::my_space
{
    vector table statements
    reserved address range
    stack definition
    heap definition
    copy table definition
    start adress
    space reference restrictions
    input section modifications
    section reference restrictions
}
```

See the subsections Stacks and heaps, Copy tables, Start address, Vector table and Reserved address ranges in Section 16.4.3, *Defining Address Spaces* for details on the keywords **stack**, **heap**, **copytable**, **vector_table** and **reserved**.

Space reference restrictions

With a space reference restriction, references from the section setup's address space to sections in specific address spaces can be deleted and blocked. If sections, for example code, in space A are not allowed or not able to access sections (functions or variables) in space B, you can configure this in LSL as follows:

```
section_setup ::A
{
    prohibit_references_to ::B;
}
```

The linker emits an error when such a reference is found in a relocation.

For example, for a multi-core AURIX, the following fragment disallows references from private0 sections to private1 and private2 sections, from private1 sections to private0 and private2 sections, and from private2 sections to private0 and private1 sections.

```
section_setup :tc0:linear
{
    prohibit_references_to :tc1:linear, :tc1:abs18, :tc2:linear, :tc2:abs18;
}

section_setup :tc1:linear
{
    prohibit_references_to :tc0:linear, :tc0:abs18, :tc2:linear, :tc2:abs18;
}

section_setup :tc2:linear
{
    prohibit_references_to :tc0:linear, :tc0:abs18, :tc1:linear, :tc1:abs18;
}
```

Input section modifications

Before sections are located and before selections defined in `section_layout` are performed, you can still modify a few section properties. These are:

• change the address space of a section

• add (`+w`) or remove (`-w`) the writable attribute

Sections are selected the same way as in groups in a `section_layout`. Instead of `attributes=+w` you can use the **copy** keyword.

```
section_setup ::A
{
    modify input (space=::B, attributes=+w)
    {
        select "mysection";
    }
}
```

Note that the new address space must be used to select a modified section in a `section_layout`. To locate the section `mysection` in the example somewhere, it must be selected in a `section_layout` for space `::B`. If the link result is output to a file, for example by only linking or incremental linking, the modified properties are exported. So, when the resulting file is used in another invocation of the linker, the section can appear in a different address space.

Section reference restrictions

With a `section_reference_restriction` statement you can specify what access operations are allowed between specific combinations of sections. Instead of selecting sections directly, the concept of a *safety class* is introduced. A safety class is associated with a collection of sections and is represented by a positive integer number. Safety classes are disjunct subsets of the set of sections that is emitted in absolute output object files. Initially, all sections are in safety class 0. By default, all access is prohibited unless the two sections are part of the same safety class. In that case, all access is allowed.

For example,

```
#define SC_ASIL_A 3
section_setup ::linear
{
    section_reference_restriction
    {
        safety_class = SC_ASIL_A;
        target_safety_class = 7;
        attributes = r;
        // safety class '3' only has read access in safety class '7'
    }
}
```

16.8. Semantics of the Section Layout Definition

Keywords in the section layout definition

```
section_layout
   direction      low_to_high  high_to_low
group
   align
   attributes     + -  r w x b i s p
   copy
   nocopy
   fill
   ordered
   contiguous
   clustered
   overlay
   allow_cross_references
   load_addr
      mem
   run_addr
```

```
      mem
   page
   page_size
   priority
   safety_class
select
stack
   size
heap
   size
reserved
   size
   attributes     r w x
   fill
   alloc_allowed absolute ranged
copytable
memcopy
   memory
   fill
section
   size
   blocksize
   attributes     r w x
   fill
   overflow
struct
   checksum

if
else
```

16.8.1. Defining a Section Layout

With the keyword **section_layout** you define a section layout for exactly one address space. In the
section layout you can specify how input sections are placed in the address space, relative to each other,
and what the absolute run and load addresses of each section will be.

You can define one or more section definitions. Each section definition arranges the sections in one
address space. You can precede the address space name with a processor name and/or core name,
separated by colons. You can omit the processor name and/or the core name if only one processor is
defined and/or only one core is present in the processor. A reference to a space in the only core of the
only processor in the system would look like "::my_space". A reference to a space of the only core on
a specific processor in the system could be "my_chip::my_space". The next example shows a section
definition for sections in the my_space address space of the processor called my_chip:

section_layout my_chip::my_space (*locate_direction*)
{
   section statements
}

Clone sections

Clone sections are placed in special address spaces that are characterized by a set of regular address spaces. Those spaces each have different "local" memory across some common address range. The clone space is created in the "main" core of the task, not in a core that is imported. For a section layout, the clone space is specified by replacing the simple space name by a list of full names for the "real" spaces. These names are separated by vertical bars (**|**), and the colons in the full names are replaced by underscores. The processor name (including the following underscore) can be omitted from the space names in the list if it is equal to the clone space's processor, and the order of spaces in the list is not important. The next example shows a definition for a multi-core TriCore with clone sections in address space `abs18`:

```
section_layout mpe:vtc:tc1_abs18|tc2_abs18|mpe_tc0_abs18
{
    section statements
}
```

You can see in the example that the `mpe_` (the processor name) can be omitted.

Locate direction

With the optional keyword **direction** you specify whether the linker starts locating sections from **low_to_high** (default) or from **high_to_low**. In the second case the linker starts locating sections at the highest addresses in the address space but preserves the order of sections when necessary (one processor and core in this example).

```
section_layout ::my_space ( direction = high_to_low )
{
    section statements
}
```

> If you do not explicitly tell the linker how to locate a section, the linker decides on the basis of the section attributes in the object file and the information in the architecture definition and memory parts where to locate the section.

16.8.2. Creating and Locating Groups of Sections

Sections are located per group. A group can contain one or more (sets of) input sections as well as other groups. Per group you can assign a mutual order to the sets of sections and locate them into a specific memory part.

```
group ( group_specifications )
{
    section_statements
}
```

With the *section_statements* you generally select sets of sections to form the group. This is described in subsection Selecting sections for a group.

Instead of selecting sections, you can also modify special sections like stack and heap or create a reserved section. This is described in Section 16.8.3, *Creating or Modifying Special Sections*.

With the *group_specifications* you actually locate the sections in the group. This is described in subsection Locating a group.

Selecting sections for a group

With the keyword **select** you can select one or more sections for the group. You can select a section by name or by attributes. If you select a section by name, you can use a wildcard pattern:

*   matches with all section names

?   matches with a single character in the section name

\   takes the next character literally

[abc]   matches with a single 'a', 'b' or 'c' character

[a-z]   matches with any single character in the range 'a' to 'z'

```
group ( ... )
{
    select "mysection";
    select "*";
}
```

The first **select** statement selects the section with the name "mysection". The second **select** statement selects all sections that were not selected yet.

A section is selected by the first select statement that matches, in the union of all section layouts for the address space. Global section layouts are processed in the order in which they appear in the LSL file. Internal core architecture section layouts always take precedence over global section layouts.

Restriction: Keep in mind that all section selections are restricted to the address space of the section layout in which this group definition occurs. So, for example for a multi-core TriCore where you have three cores, tc0, tc1 and tc2, that are imported into a virtual core vtc, when you have a section called .text.private0.lmu.p0_in_lmu in core tc0, you can select this section in the section layout for core tc0 only, so not in the root core vtc.

```
section_layout mpe:tc0:code      // must be core tc0; vtc is not allowed here
{
    group LMU_code0 ( run_addr = mem:mpe:lmuram )
    {
        select ".text.*.lmu.*"; // select section from tc0
    }
}
```

- The **attributes** field selects all sections that carry (or do not carry) the given attribute. With **+***attribute* you select sections that have the specified attribute set. With **-***attribute* you select sections that do not have the specified attribute set. You can specify one or more of the following attributes:

  - **r** readable sections

- **w** writable sections

- **x** executable sections

- **i** initialized sections

- **b** sections that should be cleared at program startup

- **s** scratch sections (not cleared and not initialized)

- **p** protected sections

To select all read-only sections:

```
group ( ... )
{
    select (attributes = +r-w);
}
```

Keep in mind that all section selections are restricted to the address space of the section layout in which this group definition occurs.

- With the **ref_tree** field you can select a group of related sections. The relation between sections is often expressed by means of references. By selecting just the 'root' of tree, the complete tree is selected. This is for example useful to locate a group of related sections in special memory (e.g. fast memory). The (referenced) sections must meet the following conditions in order to be selected:

  1. The sections are within the section layout's address space

  2. The sections match the specified attributes

  3. The sections have no absolute restriction (as is the case for all wildcard selections)

  For example, to select the code sections referenced from `foo1`:

  ```
  group refgrp (ordered, contiguous, run_addr=mem:ext_c)
  {
      select ref_tree "foo1" (attributes=+x);
  }
  ```

  If section `foo1` references `foo2` and `foo2` references `foo3`, then all these sections are selected by the selection shown above.

Locating a group

```
group group_name ( group_specifications )
{
    section_statements
}
```

With the `group_specifications` you actually define how the linker must locate the group. You can roughly define three things: 1) assign properties to the sections in a group like alignment and read/write

attributes, 2) define the mutual order in the address space for sections in the group and 3) restrict the possible addresses for the sections in a group.

The linker creates labels that allow you to refer to the begin and end address of a group from within the application software. Labels **_lc_gb_**_group_name_ and **_lc_ge_**_group_name_ mark the begin and end of the group respectively, where the begin is the lowest address used within this group and the end is the highest address used. Notice that a group not necessarily occupies all memory between begin and end address. The given label refers to where the section is located at run-time (versus load-time).

1. Assign properties to the sections in a group like alignment and read/write attributes.

   These properties are assigned to all sections in the group (and subgroups) and override the attributes of the input sections.

   - The **align** field tells the linker to align all sections in the group according to the align value. The alignment of a section is first determined by its own initial alignment and the defined alignment for the address space. Alignments are never decreased, if multiple alignments apply to a section, the largest one is used.

   - The **attributes** field tells the linker to assign one or more attributes to all sections in the group. This overrules the default attributes. By default the linker uses the attributes of the input sections. You can set the **r**, **w**, or **rw** attributes and you can switch between the **b** and **s** attributes.

   - The **copy** field tells the linker to locate a read-only section in RAM and generate a ROM copy and a copy action in the copy table. This property makes the sections in the group writable which causes the linker to generate ROM copies for the sections.

   - The effect of the **nocopy** field is the opposite of the **copy** field. It prevents the linker from generating ROM copies of the selected sections.

2. Define the mutual order of the sections in the group.

   By default, a group is *unrestricted* which means that the linker has total freedom to place the sections of the group in the address space.

   - The **ordered** keyword tells the linker to locate the sections in the same order in the address space as they appear in the group (but not necessarily adjacent).

     Suppose you have an ordered group that contains the sections 'A', 'B' and 'C'. By default the linker places the sections in the address space like 'A' - 'B' - 'C', where section 'A' gets the lowest possible address. With **direction=high_to_low** in the **section_layout** space properties, the linker places the sections in the address space like 'C' - 'B' - 'A', where section 'A' gets the highest possible address.

   - The **contiguous** keyword tells the linker to locate the sections in the group in a single address range. Within a contiguous group the input sections are located in arbitrary order, however the group occupies one contiguous range of memory. Due to alignment of sections there can be 'alignment gaps' between the sections.

     When you define a group that is both **ordered** and **contiguous**, this is called a *sequential* group. In a sequential group the linker places sections in the same order in the address space as they appear in the group and it occupies a contiguous range of memory.

- The **clustered** keyword tells the linker to locate the sections in the group in a number of *contiguous* blocks. It tries to keep the number of these blocks to a minimum. If enough memory is available, the group will be located as if it was specified as **contiguous**. Otherwise, it gets split into two or more blocks.

  If a contiguous or clustered group contains *alignment gaps*, the linker can locate sections that are not part of the group in these gaps. To prevent this, you can use the **fill** keyword. If the group is located in RAM, the gaps are treated as reserved (scratch) space. If the group is located in ROM, the alignment gaps are filled with zeros by default. You can however change the fill pattern by specifying a bit pattern. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU.

- The **overlay** keyword tells the linker to overlay the sections in the group. The linker places all sections in the address space using a contiguous range of addresses. (Thus an overlay group is automatically also a contiguous group.) To overlay the sections, all sections in the overlay group share the same run-time address.

  For each input section within the overlay the linker automatically defines two symbols. The symbol **_lc_cb_***section_name* is defined as the load-time start address of the section. The symbol **_lc_ce_***section_name* is defined as the load-time end address of the section. C (or assembly) code may be used to copy the overlaid sections.

  If sections in the overlay group contain references between groups, the linker reports an error. The keyword **allow_cross_references** tells the linker to accept cross-references. Normally, it does not make sense to have references between sections that are overlaid.

```
group ovl (overlay)
{
    group a
    {
        select "my_ovl_p1";
        select "my_ovl_p2";
    }
    group b
    {
        select "my_ovl_q1";
    }
}
```

> It may be possible that one of the sections in the overlay group already has been defined in another group where it received a load-time address. In this case the linker does not overrule this load-time address and excludes the section from the overlay group.

3. Restrict the possible addresses for the sections in a group.

   The load-time address specifies where the group's elements are loaded in memory at download time. The run-time address specifies where sections are located at run-time, that is when the program is executing. If you do not explicitly restrict the address in the LSL file, the linker assigns addresses to the sections based on the restrictions relative to other sections in the LSL file and section alignments.

The program is responsible for copying overlay sections at appropriate moment from its load-time location to its run-time location (this is typically done by the startup code).

- The **run_addr** keyword defines the run-time address. If the run-time location of a group is set explicitly, the given order between groups specify whether the run-time address propagates to the parent group or not. The location of the sections in a group can be restricted either to a single absolute address, or to a number of address ranges (not including the end address). With an expression you can specify that the group should be located at the absolute address specified by the expression:

```
group (run_addr = 0xa00f0000)
```

If the group is ordered, the first section in the group is located at the specified absolute address.

You can use the '[*offset*]' variant to locate the group at the given absolute offset in memory:

```
group (run_addr = mem:A[0x1000])
```

If the group is ordered, the first section in the group is located at the specified absolute offset in memory.

A range can be an absolute space address range, written as **[** *expr .. expr* **]**, a complete memory device, written as **mem:***mem_name*, or a memory address range, **mem:**mem_name**[***expr .. expr* **]**

```
group (run_addr = mem:my_dram)
```

You can use the '|' to specify an address range of more than one physical memory device:

```
group (run_addr = mem:A | mem:B)
```

When used in top-level section layouts, a memory name refers to a board-level memory. You can select on-chip memory with **mem:***proc_name***:***mem_name*. If the memory has multiple parallel mappings towards the current address space, you can select a specific named mapping in the memory by appending */map_name* to the memory specifier. The linker then maps memory offsets only through that mapping, so the address(es) where the sections in the group are located are determined by that memory mapping.

```
group (run_addr = mem:CPU1:A/cached)
```

- The **load_addr** keyword changes the meaning of the section selection in the group: the linker selects the load-time ROM copy of the named section(s) instead of the regular sections. Just like **run_addr** you can specify an absolute address or an address range.

```
group (contiguous, load_addr)
{
  select "mydata";  // select ROM copy of mydata:
                    // "[mydata]"
}
```

The load-time and run-time addresses of a group cannot be set at the same time. If the load-time property is set for a group, the group (only) restricts the positioning at load-time of the group's sections. It is not possible to set the address of a group that has a not-unrestricted parent group.

The properties of the load-time and run-time start address are:

- At run-time, before using an element in an overlay group, the application copies the sections from their load location to their run-time location, but only if these two addresses are different. For non-overlay sections this happens at program start-up.

- The start addresses cannot be set to absolute values for unrestricted groups.

- For non-overlay groups that do not have an overlay parent, the load-time start address equals the run-time start address.

- For any group, if the run-time start address is not set, the linker selects an appropriate address.

- If an ordered group or sequential group has an absolute address and contains sections that have separate page restrictions (not defined in LSL), all those sections are located in a single page. In other cases, for example when an unrestricted group has an address range assigned to it, the paged sections may be located in different pages.

For overlays, the linker reserves memory at the run-time start address as large as the largest element in the overlay group.

- The **page** keyword tells the linker to place the group in one page. Instead of specifying a run-time address, you can specify a page and optional a page number. Page numbers start from zero. If you omit the page number, the linker chooses a page.

  The **page** keyword refers to pages in the address space as defined in the architecture definition.

- With the **page_size** keyword you can override the page alignment and size set on the address space. When you set the page size to zero, the linker removes simple (auto generated) page restrictions from the selected sections. See also the **page_size** keyword in Section 16.4.3, *Defining Address Spaces*.

- With the **priority** keyword you can change the order in which sections are located. This is useful when some sections are considered important for good performance of the application and a small amount of fast memory is available. The value is a number for which the default is 1, so higher priorities start at 2. Sections with a higher priority are located before sections with a lower priority, unless their relative locate priority is already determined by other restrictions like **run_addr** and **page**.

```
group (priority=2)
{
  select "importantcode1";
  select "importantcode2";
}
```

- With the **safety_class** keyword you can assign a safety class to the sections in the group. The safety class must be a positive integer value. The default value is zero. The meaning of the safety class value is user defined.

```
group ASIL_A (safety_class=3)
{
```

```
    select "*.ASIL_A.*" (attributes=-w);
}
```

16.8.3. Creating or Modifying Special Sections

Instead of selecting sections, you can also create a reserved section or an output section or modify special sections like a stack or a heap. Because you cannot define these sections in the input files, you must use the linker to create them.

Stack

*   The keyword **stack** tells the linker to reserve memory for the stack. The name for the stack section refers to the stack as defined in the architecture definition. If no name was specified in the architecture definition, the default name is stack.

    With the keyword **size** you can specify the size for the stack. If the size is not specified, the linker uses the size given by the **min_size** argument as defined for the stack in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the keyword **fixed**.

    ```
    group ( ... )
    {
        stack "mystack" ( size = 2k );
    }
    ```

    The linker creates two labels to mark the begin and end of the stack, **_lc_ub_**$stack\_name$ for the begin of the stack and **_lc_ue_**$stack\_name$ for the end of the stack. The linker allocates space for the stack when there is a reference to either of the labels.

    See also the **stack** keyword in Section 16.4.3, *Defining Address Spaces*.

Heap

*   The keyword **heap** tells the linker to reserve a dynamic memory range for the malloc() function. Each heap section has a name. With the keyword **size** you can change the size for the heap. If the **size** is not specified, the linker uses the size given by the **min_size** argument as defined for the heap in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the keyword **fixed**.

    ```
    group ( ... )
    {
        heap "myheap" ( size = 2k );
    }
    ```

    The linker creates two labels to mark the begin and end of the heap, **_lc_ub_**$heap\_name$ for the begin of the heap and **_lc_ue_**$heap\_name$ for the end of the heap. The linker allocates space for the heap when a reference to either of the section labels exists in one of the input object files.

Reserved section

- The keyword **reserved** tells the linker to create an area or section of a given size. The linker will not locate any other sections in the memory occupied by a reserved section, with some exceptions. Each reserved section has a name. With the keyword **size** you can specify a size for a given reserved area or section.

```
group ( ... )
{
    reserved "myreserved" ( size = 2k );
}
```

The optional **fill** field contains a bit pattern that the linker writes to all memory addresses that remain unoccupied during the locate process. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU. The first MAU of the fill pattern is always the first MAU in the section.

By default, no sections can overlap with a reserved section. With **alloc_allowed=absolute** sections that are located at an absolute address due to an absolute group restriction can overlap a reserved section. The same applies for reserved sections with **alloc_allowed=ranged** set. Sections restricted to a fixed address range can also overlap a reserved section.

With the **attributes** field you can set the access type of the reserved section. The linker locates the reserved section in its space with the restrictions that follow from the used attributes, **r**, **w** or **x** or a valid combination of them. The allowed attributes are shown in the following table. A value between < and > in the table means this value is set automatically by the linker.

Properties set in LSL		Resulting section properties		
attributes	filled	access	memory	content
x	yes		<rom>	executable
r	yes	r	<rom>	data
r	no	r	<rom>	scratch
rx	yes	r	<rom>	executable
rw	yes	rw	<ram>	data
rw	no	rw	<ram>	scratch
rwx	yes	rw	<ram>	executable

```
group ( ... )
{
    reserved "myreserved" ( size = 2k,
              attributes = rw, fill = 0xaa );
}
```

If you do not specify any attributes, the linker will reserve the given number of maus, no matter what type of memory lies beneath. If you do not specify a fill pattern, no section is generated.

The linker creates two labels to mark the begin and end of the section, **_lc_ub_***name* for the begin of the section and **_lc_ue_***name* for the end of the reserved section.

Output sections

- The keyword `section` tells the linker to accumulate sections obtained from object files ("input sections") into an output section of a fixed size in the locate phase. You can select the input sections with `select` statements. You can use groups inside output sections, but you can only set the `align`, `attributes`, `copy` and `load_addr` properties and the `load_addr` property cannot have an address specified.

  The `fill` field contains a bit pattern that the linker writes to all unused space in the output section. When all input sections have an image (code/data) you must specify a fill pattern. If you do not specify a fill pattern, all input sections must be scratch sections. The fill pattern is aligned at the start of the output section.

  As with a reserved section you can use the `attributes` field to set the access type of the output section.

```
group ( ... )
{
    section "myoutput" ( size = 4k, attributes = rw,
                         fill = 0xaa )
    {
       select "myinput1";
       select "myinput2";
    }
}
```

  The available room for input sections is determined by the `size`, `blocksize` and `overflow` fields. With the keyword `size` you specify the fixed size of the output section. Input sections are placed from output section start towards higher addresses (offsets). When the end of the output section is reached and one or more input sections are not yet placed, an error is emitted. If however, the `overflow` field is set to another output section, remaining sections are located as if they were selected for the overflow output section.

```
group ( ... )
{
  section "tsk1_data" (size=4k, attributes=rw, fill=0,
                       overflow = "overflow_data")
  {
         select ".data.tsk1.*"
  }
  section "tsk2_data" (size=4k, attributes=rw, fill=0,
                       overflow = "overflow_data")
  {
         select ".data.tsk2.*"
  }
  section "overflow_data" (size=4k, attributes=rx,
                           fill=0)
  {
  }
}
```

With the keyword **blocksize** , the size of the output section will adapt to the size of its content. For example:

```
group flash_area (run_addr = 0x10000)
{
    section "flash_code" (blocksize=4k, attributes=rx,
                          fill=0)
    {
      select "*.flash";
    }
}
```

If the content of the section is 1 mau, the size will be 4 kB, if the content is 11 kB, the section will be 12 kB, etc. If you use **size** in combination with **blocksize**, the **size** value is used as default (minimal) size for this section. If it is omitted, the default size will be of **blocksize**. It is not allowed to omit both **size** and **blocksize** from the section definition.

The linker creates two labels to mark the begin and end of the section, **_lc_ub_***name* for the begin of the section and **_lc_ue_***name* for the end of the output section.

When the **copy** property is set on an enclosing group, a ROM copy is created for the output section and the output section itself is made writable causing it to be located in RAM by default. For this to work, the output section and its input sections must be read-only and the output section must have a **fill** property.

Copy table

- The keyword **copytable** tells the linker to select a section that is used as *copy table*. The content of the copy table is created by the linker. It contains the start address and length of all sections that should be initialized by the startup code.

  The linker creates two labels to mark the begin and end of the section, **_lc_ub_table** for the begin of the section and **_lc_ue_table** for the end of the copy table. The linker generates a copy table when a reference to either of the section labels exists in one of the input object files.

Memory copy sections

- If a memory (usually RAM) needs to be initialized by a different core than the one(s) that will use it, a copy of the contents of the memory can be placed in a section using a **memcopy** statement in a **section_layout**. All data (including code) present in the specified memory is then placed in a new section with the provided name and appropriate attributes. Unused areas in the memory are filled in the section using the supplied fill pattern or with zeros if no fill pattern is specified. If the memory contains a memory copy section the result is undefined. The actual initialization of the memory at run-time needs to be done separately, this LSL feature only directs the linker to make the data located in the memory available for initialization. Note that a memory of type **ram** cannot hold initialized data, use type **blockram** instead.

Structures

- A **struct** statement in a **section_layout** creates a section and fills it with numbers that each occupy one or more MAUs. The new section must be named by providing a double-quoted string after the

struct keyword. Each element has the form *expr*: *number*;, where the expression provides the value to insert in the section and the number determines the number of MAUs occupied by the expression value. Elements are placed in the section in the order in which they appear in the **struct** body without any gaps between them. Multi-MAU elements are split into MAUs according to the endianness of the target. A **struct** section is read-only and it cannot be copied to RAM at startup (using the **copy** group attribute). No default alignment is set.

For example,

```
struct "mystruct"
{
  0x1234                                                    : 2;
  __BMHD_ALIGN( addressof( mem:foo ), -4 )                  : 4;
  __BMHD_ALIGN( addressof( mem:foo ) + sizeof( mem:foo ), 4 )   : 4;
  checksum( crc32w,
    __BMHD_ALIGN( addressof( mem:foo ), -4 ),
    __BMHD_ALIGN( addressof( mem:foo ) + sizeof( mem:foo ), 4 ) ) : 4}
}
```

16.8.4. Creating Symbols

You can tell the linker to create symbols before locating by putting assignments in the section layout definition. Symbol names are represented by double-quoted strings. Any string is allowed, but object files may not support all characters for symbol names. You can use two different assignment operators. With the simple assignment operator '**=**', the symbol is created unconditionally. With the '**:=**' operator, the symbol is only created if it already exists as an undefined reference in an object file.

The expression that represents the value to assign to the symbol may contain references to other symbols. If such a referred symbol is a special section symbol, creation of the symbol in the left hand side of the assignment will cause creation of the special section.

```
section_layout
{
    "_lc_cp" := "_lc_ub_table";
     // when the symbol _lc_cp occurs as an undefined reference
     // in an object file, the linker generates a copy table
}
```

16.8.5. Conditional Group Statements

Within a group, you can conditionally select sections or create special sections.

- With the if keyword you can specify a condition. The succeeding section statement is executed if the condition evaluates to TRUE (1).

- The optional else keyword is followed by a section statement which is executed in case the if-condition evaluates to FALSE (0).

```
group ( ... )
{
```

```
    if ( exists( "mysection" ) )
        select "mysection";
    else
        reserved "myreserved" ( size=2k );
}
```

Chapter 17. Debug Target Configuration Files

DTC files (Debug Target Configuration files) define all possible configurations for a debug target. A debug target can be target hardware such as an evaluation board or a simulator. The DTC files are used by Eclipse to configure the project and the debugger. The information is used by the Target Board Configuration wizard and the debug configuration. DTC files are located in the `etc` directory of the installed product and use `.dtc` as filename suffix.

Based on the DTC files, the Target Board Configuration wizard adjust the project's LSL file and creates a debug launch configuration.

17.1. Custom Board Support

When you need support for a custom board and the board requires a different configuration than those that are in the product, it is necessary to create a dedicated DTC file.

To add a custom board

1. From the `etc` directory of the product, make a copy of a `.dtc` file and put it in your project directory (in the current workspace).

   In Eclipse, the DTC file should now be visible as part of your project.

2. Edit the file and give it a name that reflects the custom board.

The Import Board Configuration wizard in Eclipse adds DTC files that are present in your current project to the list of available target boards.

Syntax of a DTC file

DTC files are XML files and use the XML Schema file `dtc.xsd`, also present in the `etc` directory of the installed product.

Inspect the DTC XML schema file `dtc.xsd` for a description of the allowed elements and the available attributes. Use a delivered `.dtc` file as a starting point for creating a custom board specification.

Basically a DTC file consists of the definition of the debug target (`debugTarget` element) which embodies one or more configurations (`configuration` element) and one or more communication methods (`communicationMethod` element). The Import Board Configuration wizard in Eclipse reflects the structure of the DTC file. The elements that determine the settings that are applied by the wizard, can be found at any level in the DTC file. The wizard will apply all elements that are within the path to the selected configuration. This is best explained by an example of a DTC file with the following basic layout:

```
debugTarget: Infineon TriBoard TC1796B
    lsl
    communicationMethod: DAS over MiniWigglerII
        lsl
        configuration: Single Chip
            lsl
```

```
communicationMethod: DAS over USB-Wiggler
    lsl
    configuration: Single Chip
        lsl
lsl
```

In this example there is an LSL element at every level. If, in the Target Board Configuration wizard in Eclipse, you set the debug target configuration to "DAS over MiniWigglerII" -> "Single Chip", the wizard puts the following LSL parts into the project's LSL file in this order:

* the lsl part under the `debugTarget` element

* the lsl part under the `communicationMethod` "DAS over MiniWigglerII" element

* the lsl part under the `configuration` "Single Chip" in the `communicationMethod` "DAS over MiniWigglerII" element

* the lsl part in the `debugTarget` element at the end of the DTC file

The same applies to all other elements that determine the underlying settings.

DTC macros in LSL

To protect the Target Board Configuration wizard from changing the LSL file, you can protect the LSL file by adding the macro `__DTC_IGNORE`. This can be useful for projects that need the same LSL file, but still need to run on different target boards.

```
#define __DTC_IGNORE
```

The following DTC macros can be present in the LSL file:

LSL Define	Description
`__DTC_IGNORE`	If defined, protects the LSL file against changes by the Target Board Configuration wizard.
`__DTC_START` `__DTC_END`	The LSL part that is between these macros can be replaced by LSL text from the DTC file. If the macros are not present in the LSL file, the Target Board Configuration wizard will add them.

17.2. Description of DTC Elements and Attributes

The following table contains a description of the DTC elements and attributes. For each element a list of allowed elements is listed and the available attributes are described.

Element / Attribute	Description	Allowed Elements
`debugTarget`	The debug target.	flashChips, lsl, communicationMethod, def, processor, resource, initialize
`name`	The name of the configuration.	
`manufacturer`	The manufacturer of the debug target.	

Element / Attribute	Description	Allowed Elements
processor	Defines a processor that can be present on the debug target. Multiple processor definitions are allowed. The user should select the actual processor on the debug target.	-
name	A descriptive name of the processor derivative.	
cpu	Defines the CPU name, as for example supplied with the option **--cpu** of the control program.	
communicationMethod	Defines a communication method. A communication method is the channel that is used to communicate with the target.	ref, resource, initialize, configuration, lsl, processor
name	A descriptive name of the communication method.	
debugInstrument	The debug instrument DLL/Shared library file to be used for this communication method. Do not supply a path or a filename suffix.	
gdiMethod	This is the method used for communication. Allowed values: rs232, tcpip, can, none	
def	Define a set of elements as a macro. The macro can be expanded using the ref element.	lsl, resource, initialize, ref, configuration, flashMonitor
id	The macro name.	
resource	Defines a resource definition that can be used by Eclipse, the debugger or by the debug instrument.	-
id	The identifier name used by the debugger or debug instrument to retrieve the value.	
value	The value assigned to the resource.	
ref	Reference to a macro defined with a def element. The elements contained in the def element with the same name will be expanded at the location of the ref. Multiple refs to the same def are allowed.	-
id	The name of the referenced macro.	
configuration	Defines a configuration.	ref, initialize, resource, lsl, flashMonitor, processor
name	The descriptive name of the configuration.	

Element / Attribute	Description	Allowed Elements
`initialize`	This element defines an initialization expression. Each initialize element contains a `resourceId` attribute. If the DI requests this resource the debugger will compose a string from all initialize elements with the same `resourceId`. This DI can use this string to initialize registers by passing it to the debugger as an expression to be evaluated.	-
`resourceId`	The name of the resource to be used.	
`name`	The name of the register to be initialized.	
`value`	When the `cstart` attribute is false, this is the value to be used, otherwise, it is the default value when using this configuration. It will be used by the startup code editor to set the default register values.	
`cstart`	A boolean value. If true the debugger should ask the C startup code editor for the value, otherwise the contents of the value attribute is used. The default value is true.	
`flashMonitor`	This element specifies the flash programming monitor to be used for this configuration.	-
`monitor`	Filename of the monitor, usually an Intel Hex or S-Record file.	
`workspaceAddress`	The address of the workspace of the flash programming monitor.	
`flashSectorBufferSize`	Specifies the buffer size for buffering a flash sector.	
`chip`	This element defines a flash chip. It must be used by the flash properties page to add it on request to the list of flash chips.	debugTarget
`vendor`	The vendor of this flash chip.	
`chip`	The name of the chip.	
`width`	The width of the chip in bits.	
`chips`	The number of chips present on the board.	
`baseAddress`	The base address of the chip.	
`chipSize`	The size of the chip in bytes.	
`flashChips`	Specify a list of flash chips that can be available on this debug target.	chip

Element / Attribute	Description	Allowed Elements
lsl	Defines LSL pieces belonging to the configuration part. The LSL text must be defined between the start and end tag of this element. All LSL texts of the active selection will be placed in the project's LSL file.	-

17.3. Special Resource Identifiers

The following resource IDs are available in the TASKING VX-toolset for TriCore:

DAS debug instrument (DI): gdi2das

Resource Name	Description	Possible Values
AccessPort	The port used to connect to the wiggler.	JTAG1, USB0
DASserver	The DAS Server used for communication.	JTAG JDRV LPT JTAG over USB Box JTAG over USB Chip
DasTimeOut	The timeout value for communication with the DAS server in milliseconds. The default is 0x4000.	
RegisterFile	The core register file that is used by the debug instrument. This is usually "regbase_f7e1.dat" or "regbase_ffff.dat", depending on the register base address.	
TerminateServer	Terminate the DAS server when the session is closed.	0, 1

17.4. Initialize Elements

The initialize elements are used to initialize SFRs at startup. This is also done using a resource of the debug instrument. The following resource Ids exist for the DAS debug instrument (gdi2das):

Resource Name	Description
einit	Initialize an SFR that is protected with the ENDINIT flag.
init	Initialize an SFR that is not protected with the ENDINIT flag.

Chapter 18. CPU Problem Bypasses and Checks

Infineon Technologies regularly publishes microcontroller errata sheets for reporting both functional problems and deviations from the electrical and timing specifications.

For some of these functional problems in the microcontroller itself, the TASKING VX-toolset for TriCore provides workarounds. In fact these are software workarounds for hardware problems.

Support to deal with CPU functional problem is provided in three areas:

- Whenever possible and relevant, compiler bypasses will modify the code in order to avoid the identified erroneous code sequences;

- The assembler gives warnings for suspicious or erroneous code sequences;

- Ready-built, 'protected' standard C libraries with bypasses for all identified TriCore CPU functional problems are included in the toolset.

This chapter lists a summary of functional problems which can be bypassed by the TASKING VX-toolset for TriCore. Please refer to the Infineon errata sheets for the CPU step you are using, to verify if you need to use one of these bypasses.

To set a CPU bypass or check

1. From the **Project** menu, select **Properties for**

   The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Processor**.

   In the right pane the Processor page appears.

3. From the **Processor selection** list, select a processor.

   *The **CPU problem bypasses and checks** box shows the available workarounds/checks available for the selected processor.*

4. (Optional) Select **Show all CPU problem bypasses and checks**.

5. Click **Select All** or select one or more individual options.

Overview of the CPU problem bypasses and checks

The following table contains an overview of the silicon bugs you can provide to the C compiler option **--silicon-bug** and the assembler option **--silicon-bug**. WA means a workaround by the compiler, assembler and/or linker, CK means a check by the compiler or assembler.

CPU Problem	Description	Compiler	Assembler	Linker	CPU
CPU TC.013	Unreliable context load/store operation following an address register load instruction	WA	CK		TC1130,TC1766, TC1792, TC1796
CPU TC.048	CPU fetches program from unexpected address	WA	CK		TC1130, TC1766, TC1792, TC1796
CPU TC.060	LD.[A,DA] followed by a dependent LD.[DA,D,W] can produce unreliable results	WA	CK		TC1130, TC1766, TC1792, TC1796
CPU TC.065	Error when unconditional loop targets unconditional jump	WA	CK		TC1130, TC1766, TC1792, TC1796
CPU TC.068	Potential PSW corruption by cancelled DVINIT instructions	WA	CK		TC1130, TC1766, TC1792, TC1796
CPU TC.069	Potential incorrect operation of RSLCX instruction	WA	CK		TC1130, TC1766, TC1792, TC1796
CPU TC.070	Error when conditional jump precedes loop instruction	WA	CK		TC1130, TC1766, TC1792, TC1796
CPU TC.071	Error when Conditional Loop targets Unconditional Loop		CK		TC1130, TC1766, TC1792, TC1796
CPU TC.072	Error when Loop Counter modified prior to Loop instruction	WA	CK		TC1130, TC1766, TC1792, TC1796
CPU TC.074	Interleaved LOOP/LOOPU instructions may cause GRWP Trap		WA		TC1130, TC1766, TC1792, TC1796
CPU TC.081	Error during Load A[10], Call / Exception Sequence		CK		TC1130, TC1766, TC1792, TC1796
CPU TC.082	Data corruption possible when Memory Load follows Context Store		CK		TC1130, TC1766, TC1792, TC1796
CPU TC.083	Interrupt may be taken following DISABLE instruction	WA	CK		TC1130, TC1766, TC1792, TC1796
CPU TC.094	Potential Performance Loss when CSA Instruction follows IP Jump	WA	CK		TC1130, TC1766, TC1792, TC1796
CPU TC.095	Incorrect Forwarding in SAT, Mixed Register Instruction Sequence	WA	CK		TC1130, TC1766, TC1792, TC1796
CPU TC.096	Error when Conditional Loop targets Single Issue Group Loop	WA	CK		TC1130, TC1766, TC1792, TC1796
CPU TC.103	Spurious parity errors can be generated	WA		WA	TC1130, TC1766

CPU Problem	Description	Compiler	Assembler	Linker	CPU
CPU TC.104	Double-word Load instructions using Circular Addressing mode can produce unreliable results	WA	CK		TC1130, TC1766, TC1792, TC1796
CPU TC.105	User / Supervisor mode not staged correctly for Store Instructions	CK	CK		TC1766, TC1767, TC1792, TC1796, TC1797
CPU TC.106	Incorrect PSW update for certain IP instructions dual-issued with MTCR PSW	CK	CK		TC1767, TC1797
CPU TC.108	Incorrect Data Size for Circular Addressing mode instructions with wrap-around	WA	CK		TC1736, TC1766, TC1767, TC1792, TC1796, TC1797
CPU TC.109	Circular Addressing Load can overtake conflicting Store in Store Buffer	WA	CK		TC1736, TC1766, TC1767, TC1792, TC1796, TC1797
CPU TC.116	Parity bit corruption may occur when loop counter is read at start of loop body	WA	CK		TC1130, TC1792, TC1796

TC1164, TC1166, TC1762, TC1764 have the same silicon bugs as the TC1766.

CPU_TC.013

Command line option

`--silicon-bug=cpu-tc013`

Description

To bypass this CPU functional problem, the C compiler generates a `NOP16` instruction if a 16-bit load/store address register instruction (instructions: `LD16.A` and `ST16.A`) is followed by a lower context load/store instruction (instructions: `LDLCX` and `STLCX`).

The assembler issues a warning if a 16-bit load/store address register instruction (instructions: `LD16.A` and `ST16.A`) is followed by a lower context load/store instruction (instructions: `LDLCX` and `STLCX`).

CPU_TC.048

Command line option

`--silicon-bug=cpu-tc048`

Description

To bypass this CPU functional problem, the C compiler generates a NOP instruction before a JI or CALLI instruction when this instruction is not directly preceded by either a NOP instruction or an integer instruction or a MAC instruction. The compiler also generates a NOP instruction before a RET and RET16 instruction if there is no or just one instruction before RET, starting from the function entry point.

The assembler issues a warning when a load address register instruction LD.A/LD.DA is being followed immediately by an indirect jump JI, JLI or indirect call CALLI instruction with the same address register as parameter, because the CPU might fetch program from an unexpected address. In case there is an IP instruction between the J and LD, then the problem arises too and the assembler issues a warning. The assembler also issues a warning when there is no or just one instruction (not a NOP instruction) between label and RET or RET16.

CPU_TC.060

Command line option

```
--silicon-bug=cpu-tc060
```

Description

To bypass this CPU functional problem, the C compiler generates a NOP instruction between an LD.A / LD.DA instruction and a following LD.W / LD.D instruction, even if an integer instruction occurs in between.

The assembler issues a warning when an LD.A / LD.DA instruction is directly followed by an LD.W / LD.D instruction, or when only an integer instruction is in between.

CPU_TC.065

Command line option

`--silicon-bug=cpu-tc065`

Description

To bypass this CPU functional problem, the C compiler inserts a NOP instruction before a jump, when a label is directly followed by an unconditional jump.

The assembler issues a warning when a label is directly followed by an unconditional jump, only when debug information is turned off.

CPU_TC.068

Command line option

```
--silicon-bug=cpu-tc068
```

Description

To bypass this CPU functional problem, the C compiler inserts a `DISABLE` and two `NOP` instructions before each `DVINIT` instruction (and if necessary an `ENABLE` after `DVINIT`).

The assembler issues a warning when a `DVINIT` instruction is not preceded by a `DISABLE` and two `NOP` instructions.

CPU_TC.069

Command line option

```
--silicon-bug=cpu-tc069
```

Description

To bypass this CPU functional problem, the C compiler inserts a NOP instruction after each RSLCX instruction.

The assembler issues a warning when an RSLCX instruction is not followed by a NOP instruction.

CPU_TC.070

Command line option

`--silicon-bug=cpu-tc070`

Description

To bypass this CPU functional problem, the C compiler inserts a NOP instruction before a loop instruction, when a conditional jump, based on the value in an address register, is directly followed by a loop instruction.

The compiler inserts two NOP instructions before a loop instruction, when a conditional jump, based on the value in a data register, is directly followed by a loop instruction.

The assembler issues a warning when a conditional jump, based on the value in an address register, is directly followed by a loop instruction.

The assembler issues a warning when a conditional jump, based on the value in a data register, is directly followed by a loop instruction or when only a single NOP instruction is in between.

CPU_TC.071

Command line option

```
--silicon-bug=cpu-tc071
```

Description

The assembler issues a warning when a label is directly followed by an unconditional loop instruction, only when debug information is turned off.

The `LOOPU` instruction is never generated by the compiler.

CPU_TC.072

Command line option

```
--silicon-bug=cpu-tc072
```

Description

To bypass this CPU functional problem, the C compiler inserts a `NOP` instruction before a loop instruction, when an instruction that updates an address register is followed by a conditional loop instruction which uses this address register.

The assembler issues a warning when an instruction that updates an address register is followed by a conditional loop instruction which uses this address register.

CPU_TC.074

Command line option

```
--silicon-bug=cpu-tc074
```

Description

The C compiler has no workaround for this problem.

To bypass this CPU functional problem, the assembler encodes the LOOPU instruction in such a way that bits 12-15 get the value 1.

CPU_TC.081

Command line option

`--silicon-bug=cpu-tc081`

Description

The C compiler has no workaround for this problem.

The assembler issues a warning when an address register load instruction, `LD.A` or `LD.DA`, targeting the `A[10]` register, is immediately followed by an operation causing a context switch.

CPU_TC.082

Command line option

```
--silicon-bug=cpu-tc082
```

Description

The assembler issues a warning when a context store operation, STUCX or STLCX, is immediately followed by a memory load operation which reads from the last double-word address written by the context store.

The STUCX and STLCX instructions are never generated by the compiler.

CPU_TC.083

Command line option

```
--silicon-bug=cpu-tc083
```

Description

To bypass this CPU functional problem, the C compiler inserts a NOP instruction after each DISABLE instruction.

The assembler issues a warning when the DISABLE instruction is not followed by a NOP instruction.

CPU_TC.094

Command line option

```
--silicon-bug=cpu-tc094
```

Description

To bypass this CPU functional problem, the C compiler inserts a NOP instruction between an IP jump and CSA list instruction.

The assembler issues a warning when an IP jump is followed by a CSA list instruction.

CPU_TC.095

Command line option

`--silicon-bug=cpu-tc095`

Description

To bypass this CPU functional problem, the C compiler inserts a NOP instruction between any SAT.B/SAT.H instruction and a following load-store instruction with a DGPR source operand (addsc.a, addsc.at, mov.a, mtcr).

The assembler issues a warning when a SAT.B/SAT.H instruction is immediately followed by a load-store instruction with a DGPR source operand (addsc.a, addsc.at, mov.a, mtcr).

CPU_TC.096

Command line option

`--silicon-bug=cpu-tc096`

Description

To bypass this CPU functional problem, the C compiler inserts `NOP` instructions for a single group loop. How many depends on the pattern of the assembler instructions.

Pattern 1:

```
label:                         label:
                                        nop ; Inserted by compiler
                                        nop ; Inserted by compiler
      loop Ax,label                     loop Ax,label
```

Pattern 2:

```
label:                         label:
      <any IP instruction>              <any IP instruction>
                                        nop ; Inserted by compiler
                                        nop ; Inserted by compiler
      loop Ax,label                     loop Ax,label
```

Pattern 3:

```
label:                         label:
      <any non IP instruction>          <any non IP instruction>
                                        nop ; Inserted by compiler
      loop Ax,label                     loop Ax,label
```

Pattern 4:

```
label:                         label:
      <any IP instruction>              <any IP instruction>
      <any LS instruction>              <any LS instruction>
                                        nop ; Inserted by compiler
      loop Ax,label                     loop Ax,label
```

The assembler issues a warning if it finds one of these patterns without NOPs.

Be careful with handwritten assembly and relative addressing against the location counter. e.g.:

```
label:
        <any IP instruction>
        <any LS instruction>
        loop Ax,*-6
```

In this case it is impossible to test if one of the patterns occur. Therefore, the assembler always issues a warning when a loop instruction is used with relative addressing.

CPU_TC.103

Command line option

```
--silicon-bug=cpu-tc103
```

Description

To bypass this CPU functional problem, the C compiler directs certain program flow instructions, such as RET, RFE, CALL and JI, running in spram (scratch pad ram) via a stub located in safe memory. In order to be able to tell the C compiler that certain code is predetermined for spram, the pragma `spram` and option **--spram** are introduced.

To bypass this CPU functional problem, the preprocessor define __CPU_TC103__ is used in the `tc*.lsl` linker script files. The linker will collect the stubs as generated by the C compiler and locate them in safe non-spram memory. Furthermore it is tested if (the start of) the interrupt and trap table are located at safe addresses.

Safe non-SPRAM addresses are defined as any address except:

```
bit [15:14] = 11b (TC1130 PMEM)
bit [14:13] = 11b (TC1762, TC1764, TC1766 PMEM)
```

CPU_TC.104

Command line option

`--silicon-bug=cpu-tc104`

Description

To bypass this CPU functional problem, the C compiler inserts a `NOP` instruction before a double-word load instruction using circular addressing mode (`LD.D` instruction).

The assembler issues a warning when a double-word load instruction using circular addressing mode (`LD.D` instruction) is not preceded by a `NOP` instruction.

CPU_TC.105

Command line option

`--silicon-bug=cpu-tc105`

Description

The C compiler has no workaround for this problem. The compiler issues a warning when an `MTCR` instruction is generated, which is not preceded by a `DSYNC` instruction.

Under some conditions the use of `MTCR` leads to errors, a `DSYNC` before the `MTCR` prevents these problems, but is in most situations not necessary.

The assembler issues a warning when an `MTCR` instruction is not preceded by a `DSYNC` instruction.

CPU_TC.106

Command line option

`--silicon-bug=cpu-tc106`

Description

The C compiler has no workaround for this problem. The compiler issues a warning when an `MTCR` instruction is generated, which is preceded by a `MUL`, `MADD`, `MSUB` or `RSTV` instruction.

Under some conditions the use of `MTCR` directly after a `MUL`/`MADD`/`MSUB`/`RSTV` instruction leads to errors, a `NOP` before the `MTCR` prevents these problems, but is in most situations not necessary.

The assembler issues a warning when an `MTCR` instruction is preceded by a `MUL`, `MADD`, `MSUB` or `RSTV` instruction.

CPU_TC.108

Command line option

```
--silicon-bug=cpu-tc108
```

Description

To bypass this CPU functional problem, the C compiler inserts a NOP instruction before a load or store instruction using a circular addressing mode.

The assembler issues a warning when a load or store instruction using a circular addressing mode is not preceded by a NOP instruction.

CPU_TC.109

Command line option

`--silicon-bug=cpu-tc109`

Description

To bypass this CPU functional problem, the C compiler inserts a `NOP` instruction before the load word instruction when a load word instruction, using circular addressing mode, is preceded by a store byte instruction, or when only a single `IP` instruction is in between such a load word instruction and a store byte instruction.

The assembler issues a warning when a load word instruction, using circular addressing mode is preceded by a store byte instruction, or when only a single `IP` instruction is in between such a load word instruction and a store byte instruction.

CPU_TC.116

Command line option

```
--silicon-bug=cpu-tc116
```

Description

To bypass this CPU functional problem, the C compiler inserts two `NOP` instructions to avoid that the loop counter is read in the first two Load-Store instructions of the loop body.

The assembler issues a warning when the first two Load-Store instructions of the loop body are not preceded by two `NOP` instructions.

Details

The problem affects the following generic code sequence:

```
    ...
loop_target_:
      < Optional IPinst >
      LSinst1
      < Optional IPinst >
      LSinst2
      ...
      ...
      loop Ax, loop_target_
      ...
```

`IPinst` is any Integer Pipeline (IP) instruction, `LSinst1` and `LSinst2` are Load-Store (LS) pipeline instructions. See *TriCore1 Architecture Manual, Volume 2* (Instruction Set for V1.3/1.3.1 Architecture), Chapter 4 *Summary Tables of LS and IP Instructions* for a complete list of IP and LS instructions.

The problem may occur when either of the first two LS instructions of a loop body (`LSinst1`, `LSinst2`) read the loop variable Ax as a source operand. The problem occurs due to a missing forwarding path between the loop counter writeback to the register file and a following instruction reading this register. In this case the data seen by the LS instruction reading Ax is that currently being written through the transparent phase of the register file, potentially leading to setup timing violations where Ax is used. Such setup timing violations may lead to data corruption or spurious parity errors from memories if Ax is used as part of a memory address calculation.

Note: loop instructions themselves are not affected by this issue, so where `LSinst1` or `LSinst2` is a loop instruction then that instruction is not affected.

Chapter 19. CERT C Secure Coding Standard

The CERT C Secure Coding Standard provides rules and recommendations for secure coding in the C programming language. The goal of these rules and recommendations is to eliminate insecure coding practices and undefined behaviors that can lead to exploitable vulnerabilities. The application of the secure coding standard will lead to higher-quality systems that are robust and more resistant to attack.

This chapter contains an overview of the CERT C Secure Coding Standard recommendations and rules that are supported by the TASKING VX-toolset.

For details see the CERT C Secure Coding Standard web site. For general information about CERT secure coding, see www.cert.org/secure-coding.

Identifiers

Each rule and recommendation is given a unique identifier. These identifiers consist of three parts:

• a three-letter mnemonic representing the section of the standard

• a two-digit numeric value in the range of 00-99

• the letter "C" indicates that this is a C language guideline

The three-letter mnemonic is used to group similar coding practices and to indicate to which category a coding practice belongs.

The numeric value is used to give each coding practice a unique identifier. Numeric values in the range of 00-29 are reserved for recommendations, while values in the range of 30-99 are reserved for rules.

C compiler invocation

With the C compiler option **--cert** you can enable one or more checks for the CERT C Secure Coding Standard recommendations/rules. With **--diag=cert** you can see a list of the available checks, or you can use a three-letter mnemonic to list only the checks in a particular category. For example, **--diag=pre** lists all supported checks in the preprocessor category.

19.1. Preprocessor (PRE)

PRE01-C Use parentheses within macros around parameter names

Parenthesize all parameter names in macro definitions to avoid precedence problems.

PRE02-C   Macro replacement lists should be parenthesized

Macro replacement lists should be parenthesized to protect any lower-precedence operators from the surrounding expression. The example below is syntactically correct, although the `!=` operator was omitted. Enclosing the constant `-1` in parenthesis will prevent the incorrect interpretation and force a compiler error:

```
#define EOF -1  // should be (-1)
int getchar(void);
void f(void)
{
  if (getchar() EOF) // != operator omitted
  {
    /* ... */
  }
}
```

PRE10-C   Wrap multi-statement macros in a do-while loop

When multiple statements are used in a macro, enclose them in a `do-while` statement, so the macro can appear safely inside `if` clauses or other places that expect a single statement or a statement block. Braces alone will not work in all situations, as the macro expansion is typically followed by a semicolon.

PRE11-C   Do not conclude a single statement macro definition with a semicolon

Macro definitions consisting of a single statement should not conclude with a semicolon. If required, the semicolon should be included following the macro expansion. Inadvertently inserting a semicolon can change the control flow of the program.

19.2. Declarations and Initialization (DCL)

DCL30-C   Declare objects with appropriate storage durations

The lifetime of an automatic object ends when the function returns, which means that a pointer to the object becomes invalid.

DCL31-C   Declare identifiers before using them

The ISO C90 standard allows implicit typing of variables and functions. Because implicit declarations lead to less stringent type checking, they can often introduce unexpected and erroneous behavior or even security vulnerabilities. The ISO C99 standard requires type identifiers and forbids implicit function declarations. For backwards compatibility reasons, the VX-toolset C compiler assumes an implicit declaration and continues translation after issuing a warning message (W505 or W535).

DCL32-C   Guarantee that mutually visible identifiers are unique

The compiler encountered two or more identifiers that are identical in the first 31 characters. The ISO C99 standard allows a compiler to ignore characters past the first 31 in an identifier. Two distinct identifiers that are identical in the first 31 characters may lead to problems when the code is ported to a different compiler.

DCL35-C   Do not invoke a function using a type that does not match the function definition

This warning is generated when a function pointer is set to refer to a function of an incompatible type. Calling this function through the function pointer will result in undefined behavior. Example:

```c
void my_function(int a);
int main(void)
{
  int (*new_function)(int a) = my_function;
  return (*new_function)(10); /* the behavior is undefined */
}
```

19.3. Expressions (EXP)

EXP01-C   Do not take the size of a pointer to determine the size of the pointed-to type

The size of the object(s) allocated by malloc(), calloc() or realloc() should be a multiple of the size of the base type of the result pointer. Therefore, the sizeof expression should be applied to this base type, and not to the pointer type.

EXP12-C   Do not ignore values returned by functions

The compiler gives this warning when the result of a function call is ignored at some place, although it is not ignored for other calls to this function. This warning will not be issued when the function result is ignored for all calls, or when the result is explicitly ignored with a (void) cast.

EXP30-C   Do not depend on order of evaluation between sequence points

Between two sequence points, an object should only be modified once. Otherwise the behavior is undefined.

EXP32-C   Do not access a volatile object through a non-volatile reference

If an attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type, the behavior is undefined.

EXP33-C   Do not reference uninitialized memory

Uninitialized automatic variables default to whichever value is currently stored on the stack or in the register allocated for the variable. Consequently, uninitialized memory can cause a program to behave in an unpredictable or unplanned manner and may provide an avenue for attack.

EXP34-C   Ensure a null pointer is not dereferenced

Attempting to dereference a null pointer results in undefined behavior, typically abnormal program termination.

EXP37-C   Call functions with the arguments intended by the API

When a function is properly declared with function prototype information, an incorrect call will be flagged by the compiler. When there is no prototype information available at the call, the compiler cannot check the number of arguments and the types of the arguments. This message is issued to warn about this situation.

EXP38-C   Do not call offsetof() on bit-field members or invalid types

The behavior of the offsetof() macro is undefined when the member designator parameter designates a bit-field.

19.4. Integers (INT)

INT30-C   Ensure that unsigned integer operations do not wrap

A constant with an unsigned integer type is truncated, resulting in a wrap-around.

INT34-C   Do not shift a negative number of bits or more bits than exist in the operand

The shift count of the shift operation may be negative or greater than or equal to the size of the left operand. According to the C standard, the behavior of such a shift operation is undefined. Make sure the shift count is in range by adding appropriate range checks.

INT35-C   Evaluate integer expressions in a larger size before comparing or assigning to that size

If an integer expression is compared to, or assigned to a larger integer size, that integer expression should be evaluated in that larger size by explicitly casting one of the operands.

19.5. Floating Point (FLP)

FLP30-C   Do not use floating point variables as loop counters

To avoid problems with limited precision and rounding, floating point variables should not be used as loop counters.

FLP35-C   Take granularity into account when comparing floating point values

Floating point arithmetic in C is inexact, so floating point values should not be tested for exact equality or inequality.

FLP36-C   Beware of precision loss when converting integral types to floating point

Conversion from integral types to floating point types without sufficient precision can lead to loss of precision.

19.6. Arrays (ARR)

ARR01-C Do not apply the sizeof operator to a pointer when taking the size of an array

A function parameter declared as an array, is converted to a pointer by the compiler. Therefore, the sizeof operator applied to this parameter yields the size of a pointer, and not the size of an array.

ARR34-C Ensure that array types in expressions are compatible

Using two or more incompatible arrays in an expression results in undefined behavior.

ARR35-C Do not allow loops to iterate beyond the end of an array

Reading or writing of data outside the bounds of an array may lead to incorrect program behavior or execution of arbitrary code.

19.7. Characters and Strings (STR)

STR30-C Do not attempt to modify string literals

Writing to a string literal has undefined behavior, as identical strings may be shared and/or allocated in read-only memory.

STR33-C Size wide character strings correctly

Wide character strings may be improperly sized when they are mistaken for narrow strings or for multi-byte character strings.

STR34-C Cast characters to unsigned types before converting to larger integer sizes

A signed character is sign-extended to a larger signed integer value. Use an explicit cast, or cast the value to an unsigned type first, to avoid unexpected sign-extension.

STR36-C Do not specify the bound of a character array initialized with a string literal

The compiler issues this warning when the character buffer initialized by a string literal does not provide enough room for the terminating null character.

19.8. Memory Management (MEM)

MEM00-C Allocate and free memory in the same module, at the same level of abstraction

The compiler issues this warning when the result of the call to malloc(), calloc() or realloc() is discarded, and therefore not free()d, resulting in a memory leak.

MEM08-C Use realloc() only to resize dynamically allocated arrays

Only use realloc() to resize an array. Do not use it to transform an object to an object of a different type.

MEM30-C   Do not access freed memory

When memory is freed, its contents may remain intact and accessible because it is at the memory manager's discretion when to reallocate or recycle the freed chunk. The data at the freed location may appear valid. However, this can change unexpectedly, leading to unintended program behavior. As a result, it is necessary to guarantee that memory is not written to or read from once it is freed.

MEM31-C   Free dynamically allocated memory exactly once

Freeing memory multiple times has similar consequences to accessing memory after it is freed. The underlying data structures that manage the heap can become corrupted. To eliminate double-free vulnerabilities, it is necessary to guarantee that dynamic memory is freed exactly once.

MEM32-C   Detect and handle memory allocation errors

The result of realloc() is assigned to the original pointer, without checking for failure. As a result, the original block of memory is lost when realloc() fails.

MEM33-C   Use the correct syntax for flexible array members

Use the ISO C99 syntax for flexible array members instead of an array member of size 1.

MEM34-C   Only free memory allocated dynamically

Freeing memory that is not allocated dynamically can lead to corruption of the heap data structures.

MEM35-C   Allocate sufficient memory for an object

The compiler issues this warning when the size of the object(s) allocated by malloc(), calloc() or realloc() is smaller than the size of an object pointed to by the result pointer. This may be caused by a sizeof expression with the wrong type or with a pointer type instead of the object type.

19.9. Environment (ENV)

ENV32-C   All atexit handlers must return normally

The compiler issues this warning when an atexit() handler is calling a function that does not return. No atexit() registered handler should terminate in any way other than by returning.

19.10. Signals (SIG)

SIG30-C   Call only asynchronous-safe functions within signal handlers

SIG32-C   Do not call longjmp() from inside a signal handler

Invoking the longjmp() function from within a signal handler can lead to undefined behavior if it results in the invocation of any non-asynchronous-safe functions, likely compromising the integrity of the program.

19.11. Miscellaneous (MSC)

MSC32-C Ensure your random number generator is properly seeded

Ensure that the random number generator is properly seeded by calling srand().

Chapter 20. MISRA C Rules

This chapter contains an overview of the supported and unsupported MISRA C rules.

20.1. MISRA C:1998

This section lists all supported and unsupported MISRA C:1998 rules.

See also Section 4.7.2, *C Code Checking: MISRA C*.

A number of MISRA C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

x means that the rule is not supported by the TASKING C compiler. (R) is a required rule, (A) is an advisory rule.

	1.	(R)	The code shall conform to standard C, without language extensions.
x	2.	(A)	Other languages should only be used with an interface standard.
	3.	(A)	Inline assembly is only allowed in dedicated C functions.
x	4.	(A)	Provision should be made for appropriate run-time checking.
	5.	(R)	Only use characters and escape sequences defined by ISO C.
x	6.	(R)	Character values shall be restricted to a subset of ISO 106460-1.
	7.	(R)	Trigraphs shall not be used.
	8.	(R)	Multibyte characters and wide string literals shall not be used.
	9.	(R)	Comments shall not be nested.
	10.	(A)	Sections of code should not be "commented out".

In general, it is not possible to decide whether a piece of comment is C code that is commented out, or just some pseudo code. Instead, the following heuristics are used to detect possible C code inside a comment:

- a line ends with ';', or

- a line starts with '}', possibly preceded by white space

	11.	(R)	Identifiers shall not rely on significance of more than 31 characters.
	12.	(A)	The same identifier shall not be used in multiple name spaces.
	13.	(A)	Specific-length typedefs should be used instead of the basic types.
	14.	(R)	Use `unsigned char` or `signed char` instead of plain `char`.
x	15.	(A)	Floating-point implementations should comply with a standard.
	16.	(R)	The bit representation of floating-point numbers shall not be used. A violation is reported when a pointer to a floating-point type is converted to a pointer to an integer type.

	17.	(R)	`typedef` names shall not be reused.
	18.	(A)	Numeric constants should be suffixed to indicate type. A violation is reported when the value of the constant is outside the range indicated by the suffixes, if any.
	19.	(R)	Octal constants (other than zero) shall not be used.
	20.	(R)	All object and function identifiers shall be declared before use.
	21.	(R)	Identifiers shall not hide identifiers in an outer scope.
	22.	(A)	Declarations should be at function scope where possible.
x	23.	(A)	All declarations at file scope should be static where possible.
	24.	(R)	Identifiers shall not have both internal and external linkage.
x	25.	(R)	Identifiers with external linkage shall have exactly one definition.
	26.	(R)	Multiple declarations for objects or functions shall be compatible.
x	27.	(A)	External objects should not be declared in more than one file.
	28.	(A)	The `register` storage class specifier should not be used.
	29.	(R)	The use of a tag shall agree with its declaration.
	30.	(R)	All automatics shall be initialized before being used . This rule is checked using worst-case assumptions. This means that violations are reported not only for variables that are guaranteed to be uninitialized, but also for variables that are uninitialized on some execution paths.
	31.	(R)	Braces shall be used in the initialization of arrays and structures.
	32.	(R)	Only the first, or all enumeration constants may be initialized.
	33.	(R)	The right hand operand of `&&` or `||` shall not contain side effects.
	34.	(R)	The operands of a logical `&&` or `||` shall be primary expressions.
	35.	(R)	Assignment operators shall not be used in Boolean expressions.
	36.	(A)	Logical operators should not be confused with bitwise operators.
	37.	(R)	Bitwise operations shall not be performed on signed integers.
	38.	(R)	A shift count shall be between 0 and the operand width minus 1. This violation will only be checked when the shift count evaluates to a constant value at compile time.
	39.	(R)	The unary minus shall not be applied to an unsigned expression.
	40.	(A)	`sizeof` should not be used on expressions with side effects.
x	41.	(A)	The implementation of integer division should be documented.
	42.	(R)	The comma operator shall only be used in a `for` condition.
	43.	(R)	Don't use implicit conversions which may result in information loss.
	44.	(A)	Redundant explicit casts should not be used.
	45.	(R)	Type casting from any type to or from pointers shall not be used.

46. (R) The value of an expression shall be evaluation order independent.
This rule is checked using worst-case assumptions. This means that a violation will be reported when a possible alias may cause the result of an expression to be evaluation order dependent.

47. (A) No dependence should be placed on operator precedence rules.

48. (A) Mixed arithmetic should use explicit casting.

49. (A) Tests of a (non-Boolean) value against 0 should be made explicit.

50. (R) F.P. variables shall not be tested for exact equality or inequality.

51. (A) Constant unsigned integer expressions should not wrap-around.

52. (R) There shall be no unreachable code.

53. (R) All non-null statements shall have a side-effect.

54. (R) A null statement shall only occur on a line by itself.

55. (A) Labels should not be used.

56. (R) The `goto` statement shall not be used.

57. (R) The `continue` statement shall not be used.

58. (R) The `break` statement shall not be used (except in a `switch`).

59. (R) An `if` or loop body shall always be enclosed in braces.

60. (A) All `if`, `else if` constructs should contain a final `else`.

61. (R) Every non-empty `case` clause shall be terminated with a `break`.

62. (R) All `switch` statements should contain a final `default` case.

63. (A) A `switch` expression should not represent a Boolean case.

64. (R) Every `switch` shall have at least one `case`.

65. (R) Floating-point variables shall not be used as loop counters.

66. (A) A `for` should only contain expressions concerning loop control.
A violation is reported when the loop initialization or loop update expression modifies an object that is not referenced in the loop test.

67. (A) Iterator variables should not be modified in a `for` loop.

68. (R) Functions shall always be declared at file scope.

69. (R) Functions with variable number of arguments shall not be used.

70. (R) Functions shall not call themselves, either directly or indirectly.
A violation will be reported for direct or indirect recursive function calls in the source file being checked. Recursion via functions in other source files, or recursion via function pointers is not detected.

71. (R) Function prototypes shall be visible at the definition and call.

72. (R) The function prototype of the declaration shall match the definition.

73. (R) Identifiers shall be given for all prototype parameters or for none.

74. (R) Parameter identifiers shall be identical for declaration/definition.

75. (R) Every function shall have an explicit return type.

76.   (R)   Functions with no parameters shall have a `void` parameter list.

77.   (R)   An actual parameter type shall be compatible with the prototype.

78.   (R)   The number of actual parameters shall match the prototype.

79.   (R)   The values returned by `void` functions shall not be used.

80.   (R)   Void expressions shall not be passed as function parameters.

81.   (A)   `const` should be used for reference parameters not modified.

82.   (A)   A function should have a single point of exit.

83.   (R)   Every exit point shall have a `return` of the declared return type.

84.   (R)   For `void` functions, `return` shall not have an expression.

85.   (A)   Function calls with no parameters should have empty parentheses.

86.   (A)   If a function returns error information, it should be tested.
A violation is reported when the return value of a function is ignored.

87.   (R)   `#include` shall only be preceded by other directives or comments.

88.   (R)   Non-standard characters shall not occur in `#include` directives.

89.   (R)   `#include` shall be followed by either `<filename>` or `"filename"`.

90.   (R)   Plain macros shall only be used for constants/qualifiers/specifiers.

91.   (R)   Macros shall not be `#define`'d and `#undef`'d within a block.

92.   (A)   `#undef` should not be used.

93.   (A)   A function should be used in preference to a function-like macro.

94.   (R)   A function-like macro shall not be used without all arguments.

95.   (R)   Macro arguments shall not contain pre-preprocessing directives.
A violation is reported when the first token of an actual macro argument is '#'.

96.   (R)   Macro definitions/parameters should be enclosed in parentheses.

97.   (A)   Don't use undefined identifiers in pre-processing directives.

98.   (R)   A macro definition shall contain at most one # or ## operator.

99.   (R)   All uses of the `#pragma` directive shall be documented.
This rule is really a documentation issue. The compiler will flag all `#pragma` directives as violations.

100.   (R)   `defined` shall only be used in one of the two standard forms.

101.   (A)   Pointer arithmetic should not be used.

102.   (A)   No more than 2 levels of pointer indirection should be used.
A violation is reported when a pointer with three or more levels of indirection is declared.

103.   (R)   No relational operators between pointers to different objects.
In general, checking whether two pointers point to the same object is impossible. The compiler will only report a violation for a relational operation with incompatible pointer types.

104.   (R)   Non-constant pointers to functions shall not be used.

105.   (R)   Functions assigned to the same pointer shall be of identical type.

| | 106. | (R) | Automatic address may not be assigned to a longer lived object. |

106. (R) Automatic address may not be assigned to a longer lived object.

107. (R) The null pointer shall not be de-referenced.
A violation is reported for every pointer dereference that is not guarded by a NULL pointer test.

108. (R) All `struct/union` members shall be fully specified.

109. (R) Overlapping variable storage shall not be used.
A violation is reported for every `union` declaration.

110. (R) Unions shall not be used to access the sub-parts of larger types.
A violation is reported for a `union` containing a `struct` member.

111. (R) Bit-fields shall have type `unsigned int` or `signed int`.

112. (R) Bit-fields of type `signed int` shall be at least 2 bits long.

113. (R) All `struct/union` members shall be named.

114. (R) Reserved and standard library names shall not be redefined.

115. (R) Standard library function names shall not be reused.

x 116. (R) Production libraries shall comply with the MISRA C restrictions.

x 117. (R) The validity of library function parameters shall be checked.

118. (R) Dynamic heap memory allocation shall not be used.

119. (R) The error indicator `errno` shall not be used.

120. (R) The macro `offsetof` shall not be used.

121. (R) `<locale.h>` and the `setlocale` function shall not be used.

122. (R) The `setjmp` and `longjmp` functions shall not be used.

123. (R) The signal handling facilities of `<signal.h>` shall not be used.

124. (R) The `<stdio.h>` library shall not be used in production code.

125. (R) The functions `atof/atoi/atol` shall not be used.

126. (R) The functions `abort/exit/getenv/system` shall not be used.

127. (R) The time handling functions of library `<time.h>` shall not be used.

20.2. MISRA C:2004

This section lists all supported and unsupported MISRA C:2004 rules.

See also Section 4.7.2, *C Code Checking: MISRA C*.

A number of MISRA C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

x means that the rule is not supported by the TASKING C compiler. (R) is a required rule, (A) is an advisory rule.

Environment

1.1   (R)   All code shall conform to ISO 9899:1990 "Programming languages - C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.

1.2   (R)   No reliance shall be placed on undefined or unspecified behavior.

x   1.3   (R)   Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform.

x   1.4   (R)   The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.

x   1.5   (A)   Floating-point implementations should comply with a defined floating-point standard.

Language extensions

2.1   (R)   Assembly language shall be encapsulated and isolated.

2.2   (R)   Source code shall only use `/* ... */` style comments.

2.3   (R)   The character sequence `/*` shall not be used within a comment.

2.4   (A)   Sections of code should not be "commented out". In general, it is not possible to decide whether a piece of comment is C code that is commented out, or just some pseudo code. Instead, the following heuristics are used to detect possible C code inside a comment: - a line ends with ';', or - a line starts with '}', possibly preceded by white space

Documentation

x   3.1   (R)   All usage of implementation-defined behavior shall be documented.

x   3.2   (R)   The character set and the corresponding encoding shall be documented.

x   3.3   (A)   The implementation of integer division in the chosen compiler should be determined, documented and taken into account.

3.4   (R)   All uses of the `#pragma` directive shall be documented and explained. This rule is really a documentation issue. The compiler will flag all `#pragma` directives as violations.

3.5   (R)   The implementation-defined behavior and packing of bit-fields shall be documented if being relied upon.

x   3.6   (R)   All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.

Character sets

4.1   (R)   Only those escape sequences that are defined in the ISO C standard shall be used.

4.2   (R)   Trigraphs shall not be used.

Identifiers

5.1	(R)	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
5.2	(R)	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
5.3	(R)	A `typedef` name shall be a unique identifier.
5.4	(R)	A tag name shall be a unique identifier.
5.5	(A)	No object or function identifier with static storage duration should be reused.
5.6	(A)	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.
5.7	(A)	No identifier name should be reused.

Types

6.1	(R)	The plain `char` type shall be used only for storage and use of character values.
6.2	(R)	`signed` and `unsigned char` type shall be used only for the storage and use of numeric values.
6.3	(A)	`typedefs` that indicate size and signedness should be used in place of the basic types.
6.4	(R)	Bit-fields shall only be defined to be of type `unsigned int` or `signed int.`
6.5	(R)	Bit-fields of type `signed int` shall be at least 2 bits long.

Constants

7.1	(R)	Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and definitions

8.1	(R)	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
8.2	(R)	Whenever an object or function is declared or defined, its type shall be explicitly stated.
8.3	(R)	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.
8.4	(R)	If objects or functions are declared more than once their types shall be compatible.
8.5	(R)	There shall be no definitions of objects or functions in a header file.
8.6	(R)	Functions shall be declared at file scope.
8.7	(R)	Objects shall be defined at block scope if they are only accessed from within a single function.
8.8	(R)	An external object or function shall be declared in one and only one file.

	8.9	(R)	An identifier with external linkage shall have exactly one external definition.
x	8.10	(R)	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.
	8.11	(R)	The `static` storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
	8.12	(R)	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

Initialization

9.1   (R)   All automatic variables shall have been assigned a value before being used. This rule is checked using worst-case assumptions. This means that violations are reported not only for variables that are guaranteed to be uninitialized, but also for variables that are uninitialized on some execution paths.

9.2   (R)   Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

9.3   (R)   In an enumerator list, the "=" construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Arithmetic type conversions

10.1   (R)   The value of an expression of integer type shall not be implicitly converted to a different underlying type if:
a) it is not a conversion to a wider integer type of the same signedness, or
b) the expression is complex, or
c) the expression is not constant and is a function argument, or
d) the expression is not constant and is a return expression.

10.2   (R)   The value of an expression of floating type shall not be implicitly converted to a different type if:
a) it is not a conversion to a wider floating type, or
b) the expression is complex, or
c) the expression is a function argument, or
d) the expression is a return expression.

10.3   (R)   The value of a complex expression of integer type may only be cast to a type of the same signedness that is no wider than the underlying type of the expression.

10.4   (R)   The value of a complex expression of floating type may only be cast to a type that is no wider than the underlying type of the expression.

10.5   (R)   If the bitwise operators `~` and `<<` are applied to an operand of underlying type `unsigned char` or `unsigned short`, the result shall be immediately cast to the underlying type of the operand.

10.6   (R)   A "U" suffix shall be applied to all constants of `unsigned` type.

Pointer type conversions

11.1 (R) Conversions shall not be performed between a pointer to a function and any type other than an integral type.

11.2 (R) Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.

11.3 (A) A cast should not be performed between a pointer type and an integral type.

11.4 (A) A cast should not be performed between a pointer to object type and a different pointer to object type.

11.5 (R) A cast shall not be performed that removes any `const` or `volatile` qualification from the type addressed by a pointer.

Expressions

12.1 (A) Limited dependence should be placed on C's operator precedence rules in expressions.

12.2 (R) The value of an expression shall be the same under any order of evaluation that the standard permits. This rule is checked using worst-case assumptions. This means that a violation will be reported when a possible alias may cause the result of an expression to be evaluation order dependent.

12.3 (R) The `sizeof` operator shall not be used on expressions that contain side effects.

12.4 (R) The right-hand operand of a logical `&&` or `||` operator shall not contain side effects.

12.5 (R) The operands of a logical `&&` or `||` shall be *primary-expressions*.

12.6 (A) The operands of logical operators (`&&`, `||` and `!`) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (`&&`, `||` and `!`).

12.7 (R) Bitwise operators shall not be applied to operands whose underlying type is signed.

12.8 (R) The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand. This violation will only be checked when the shift count evaluates to a constant value at compile time.

12.9 (R) The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

12.10 (R) The comma operator shall not be used.

12.11 (A) Evaluation of constant unsigned integer expressions should not lead to wrap-around.

12.12 (R) The underlying bit representations of floating-point values shall not be used. A violation is reported when a pointer to a floating-point type is converted to a pointer to an integer type.

12.13 (A) The increment (`++`) and decrement (`--`) operators should not be mixed with other operators in an expression.

Control statement expressions

13.1 (R) Assignment operators shall not be used in expressions that yield a Boolean value.

13.2   (A)   Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.

13.3   (R)   Floating-point expressions shall not be tested for equality or inequality.

13.4   (R)   The controlling expression of a `for` statement shall not contain any objects of floating type.

13.5   (R)   The three expressions of a `for` statement shall be concerned only with loop control. A violation is reported when the loop initialization or loop update expression modifies an object that is not referenced in the loop test.

13.6   (R)   Numeric variables being used within a `for` loop for iteration counting shall not be modified in the body of the loop.

13.7   (R)   Boolean operations whose results are invariant shall not be permitted.

Control flow

14.1   (R)   There shall be no unreachable code.

14.2   (R)   All non-null statements shall either:
a) have at least one side effect however executed, or
b) cause control flow to change.

14.3   (R)   Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.

14.4   (R)   The `goto` statement shall not be used.

14.5   (R)   The `continue` statement shall not be used.

14.6   (R)   For any iteration statement there shall be at most one break statement used for loop termination.

14.7   (R)   A function shall have a single point of exit at the end of the function.

14.8   (R)   The statement forming the body of a `switch, while, do ... while` or `for` statement be a compound statement.

14.9   (R)   An `if (`*expression*`)` construct shall be followed by a compound statement. The `else` keyword shall be followed by either a compound statement, or another `if` statement.

14.10 (R)   All `if ... else if` constructs shall be terminated with an `else` clause.

Switch statements

15.1   (R)   A switch label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement.

15.2   (R)   An unconditional `break` statement shall terminate every non-empty `switch` clause.

15.3   (R)   The final clause of a switch statement shall be the `default` clause.

15.4   (R)   A `switch` expression shall not represent a value that is effectively Boolean.

15.5   (R)   Every `switch` statement shall have at least one `case` clause.

Functions

16.1 (R)   Functions shall not be defined with variable numbers of arguments.

16.2 (R)   Functions shall not call themselves, either directly or indirectly. A violation will be reported for direct or indirect recursive function calls in the source file being checked. Recursion via functions in other source files, or recursion via function pointers is not detected.

16.3 (R)   Identifiers shall be given for all of the parameters in a function prototype declaration.

16.4 (R)   The identifiers used in the declaration and definition of a function shall be identical.

16.5 (R)   Functions with no parameters shall be declared with parameter type `void`.

16.6 (R)   The number of arguments passed to a function shall match the number of parameters.

16.7 (A)   A pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.

16.8 (R)   All exit paths from a function with non-void return type shall have an explicit `return` statement with an expression.

16.9 (R)   A function identifier shall only be used with either a preceding `&`, or with a parenthesized parameter list, which may be empty.

16.10 (R)   If a function returns error information, then that error information shall be tested. A violation is reported when the return value of a function is ignored.

Pointers and arrays

x   17.1 (R)   Pointer arithmetic shall only be applied to pointers that address an array or array element.

x   17.2 (R)   Pointer subtraction shall only be applied to pointers that address elements of the same array.

17.3 (R)   `>`, `>=`, `<`, `<=` shall not be applied to pointer types except where they point to the same array. In general, checking whether two pointers point to the same object is impossible. The compiler will only report a violation for a relational operation with incompatible pointer types.

17.4 (R)   Array indexing shall be the only allowed form of pointer arithmetic.

17.5 (A)   The declaration of objects should contain no more than 2 levels of pointer indirection. A violation is reported when a pointer with three or more levels of indirection is declared.

17.6 (R)   The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Structures and unions

18.1 (R)   All structure or union types shall be complete at the end of a translation unit.

18.2 (R)   An object shall not be assigned to an overlapping object.

x   18.3 (R)   An area of memory shall not be reused for unrelated purposes.

18.4 (R) Unions shall not be used.

Preprocessing directives

19.1 (A) `#include` statements in a file should only be preceded by other preprocessor directives or comments.

19.2 (A) Non-standard characters should not occur in header file names in `#include` directives.

x 19.3 (R) The `#include` directive shall be followed by either a `<filename>` or `"filename"` sequence.

19.4 (R) C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.

19.5 (R) Macros shall not be `#define`'d or `#undef`'d within a block.

19.6 (R) `#undef` shall not be used.

19.7 (A) A function should be used in preference to a function-like macro.

19.8 (R) A function-like macro shall not be invoked without all of its arguments.

19.9 (R) Arguments to a function-like macro shall not contain tokens that look like preprocessing directives. A violation is reported when the first token of an actual macro argument is '#'.

19.10 (R) In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.

19.11 (R) All macro identifiers in preprocessor directives shall be defined before use, except in `#ifdef` and `#ifndef` preprocessor directives and the `defined()` operator.

19.12 (R) There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.

19.13 (A) The # and ## preprocessor operators should not be used.

19.14 (R) The `defined` preprocessor operator shall only be used in one of the two standard forms.

19.15 (R) Precautions shall be taken in order to prevent the contents of a header file being included twice.

19.16 (R) Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.

19.17 (R) All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` directive to which they are related.

Standard libraries

20.1 (R) Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.

20.2 (R) The names of standard library macros, objects and functions shall not be reused.

x 20.3 (R) The validity of values passed to library functions shall be checked.

20.4 (R) Dynamic heap memory allocation shall not be used.

20.5 (R) The error indicator `errno` shall not be used.

20.6 (R) The macro `offsetof`, in library `<stddef.h>`, shall not be used.

20.7 (R) The `setjmp` macro and the `longjmp` function shall not be used.

20.8 (R) The signal handling facilities of `<signal.h>` shall not be used.

20.9 (R) The input/output library `<stdio.h>` shall not be used in production code.

20.10 (R) The library functions `atof`, `atoi` and `atol` from library `<stdlib.h>` shall not be used.

20.11 (R) The library functions `abort`, `exit`, `getenv` and `system` from library `<stdlib.h>` shall not be used.

20.12 (R) The time handling functions of library `<time.h>` shall not be used.

Run-time failures

x 21.1 (R) Minimization of run-time failures shall be ensured by the use of at least one of:
a) static analysis tools/techniques;
b) dynamic analysis tools/techniques;
c) explicit coding of checks to handle run-time faults.

20.3. MISRA C:2012

This section lists all supported and unsupported MISRA C:2012 rules.

See also Section 4.7.2, *C Code Checking: MISRA C*.

A number of MISRA C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

x means that the rule is not supported by the TASKING C compiler. (M) is a mandatory rule, (R) is a required rule, (A) is an advisory rule.

A standard C environment

1.1 (R) The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.

1.2 (A) Language extensions should not be used.

1.3 (R) There shall be no occurrence of undefined or critical unspecified behavior.

Unused code

2.1 (R) A project shall not contain unreachable code.

2.2 (R) There shall be no dead code.

2.3 (A) A project should not contain unused type declarations.

2.4 (A) A project should not contain unused tag declarations.

2.5 (A) A project should not contain unused macro declarations.

2.6 (A) A function should not contain unused label declarations.

2.7 (A) There should be no unused parameters in functions.

Comments

3.1 (R) The character sequences `/*` and `//` shall not be used within a comment.

3.2 (R) Line-splicing shall not be used in `//` comments.

Character sets and lexical conventions

4.1 (R) Octal and hexadecimal escape sequences shall be terminated.

4.2 (A) Trigraphs should not be used.

Identifiers

x 5.1 (R) External identifiers shall be distinct.

x 5.2 (R) Identifiers declared in the same scope and name space shall be distinct.

x 5.3 (R) An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.

x 5.4 (R) Macro identifiers shall be distinct.

x 5.5 (R) Identifiers shall be distinct from macro names.

x 5.6 (R) A `typedef` name shall be a unique identifier.

x 5.7 (R) A tag name shall be a unique identifier.

x 5.8 (R) Identifiers that define objects or functions with external linkage shall be unique.

x 5.9 (A) Identifiers that define objects or functions with internal linkage should be unique.

Types

6.1 (R) Bit-fields shall only be declared with an appropriate type.

6.2 (R) Single-bit named bit-fields shall not be of a signed type.

Literals and constants

7.1 (R) Octal constants shall not be used.

7.2 (R) A "`u`" or "`U`" suffix shall be applied to all integer constants that are represented in an `unsigned` type.

7.3 (R) The lowercase character "`l`" shall not be used in a literal suffix trivial.

7.4 (R) A string literal shall not be assigned to an object unless the object's type is "pointer to `const`-qualified `char`".

Declarations and definitions

8.1	(R)	Types shall be explicitly specified.
8.2	(R)	Function types shall be in prototype form with named parameters.
8.3	(R)	All declarations of an object or function shall use the same names and type qualifiers.
8.4	(R)	A compatible declaration shall be visible when an object or function with external linkage is defined.
8.5	(R)	An external object or function shall be declared once in one and only one file.
8.6	(R)	An identifier with external linkage shall have exactly one external definition.
8.7	(A)	Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.
8.8	(R)	The `static` storage class specifier shall be used in all declarations of objects and functions that have internal linkage.
8.9	(A)	An object should be defined at block scope if its identifier only appears in a single function.
8.10	(R)	An inline function shall be declared with the `static` storage class.
8.11	(A)	When an array with external linkage is declared, its size should be explicitly specified.
8.12	(R)	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.
8.13	(A)	A pointer should point to a `const`-qualified type whenever possible.
8.14	(R)	The `restrict` type qualifier shall not be used.

Initialization

9.1	(M)	The value of an object with automatic storage duration shall not be read before it has been set.
9.2	(R)	The initializer for an aggregate or union shall be enclosed in braces.
9.3	(R)	Arrays shall not be partially initialized.
9.4	(R)	An element of an object shall not be initialized more than once.
9.5	(R)	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

The essential type model

10.1	(R)	Operands shall not be of an inappropriate essential type.
10.2	(R)	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.
10.3	(R)	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.
10.4	(R)	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.

10.5 (A) The value of an expression should not be cast to an inappropriate essential type.

10.6 (R) The value of a composite expression shall not be assigned to an object with wider essential type.

10.7 (R) If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.

10.8 (R) The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

Pointer type conversions

11.1 (R) Conversions shall not be performed between a pointer to a function and any other type.

11.2 (R) Conversions shall not be performed between a pointer to an incomplete type and any other type.

11.3 (R) A cast shall not be performed between a pointer to object type and a pointer to a different object type.

11.4 (A) A conversion should not be performed between a pointer to object and an integer type.

11.5 (A) A conversion should not be performed from pointer to `void` into pointer to object.

11.6 (R) A cast shall not be performed between pointer to `void` and an arithmetic type.

11.7 (R) A cast shall not be performed between pointer to object and a non-integer arithmetic type.

11.8 (R) A cast shall not remove any `const` or `volatile` qualification from the type pointed to by a pointer.

11.9 (R) The macro `NULL` shall be the only permitted form of integer null pointer constant.

Expressions

12.1 (A) The precedence of operators within expressions should be made explicit.

12.2 (R) The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand.

12.3 (A) The comma operator should not be used.

12.4 (A) Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Side effects

13.1 (R) Initializer lists shall not contain persistent side effects.

13.2 (R) The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.

13.3 (A) A full expression containing an increment (`++`) or decrement (`--`) operator should have no other potential side effects other than that caused by the increment or decrement operator.

13.4 (A) The result of an assignment operator should not be used.

13.5 (R) The right hand operand of a logical `&&` or `||` operator shall not contain persistent side effects.

13.6 (M) The operand of the `sizeof` operator shall not contain any expression which has potential side effects.

Control statement expressions

14.1 (R) A loop counter shall not have essentially floating type.

14.2 (R) A `for` loop shall be well-formed.

14.3 (R) Controlling expressions shall not be invariant.

14.4 (R) The controlling expression of an `if` statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

Control flow

15.1 (A) The `goto` statement should not be used.

15.2 (R) The `goto` statement shall jump to a label declared later in the same function.

15.3 (R) Any label referenced by a `goto` statement shall be declared in the same block, or in any block enclosing the `goto` statement.

15.4 (A) There should be no more than one `break` or `goto` statement used to terminate any iteration statement.

15.5 (A) A function should have a single point of exit at the end.

15.6 (R) The body of an iteration-statement or a selection-statement shall be a compound-statement.

15.7 (R) All `if ... else if` constructs shall be terminated with an `else` statement.

Switch statements

16.1 (R) All `switch` statements shall be well-formed.

16.2 (R) A switch label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement.

16.3 (R) An unconditional `break` statement shall terminate every switch-clause.

16.4 (R) Every `switch` statement shall have a `default` label.

16.5 (R) A `default` label shall appear as either the first or the last switch label of a `switch` statement.

16.6 (R) Every `switch` statement shall have at least two switch-clauses.

16.7 (R) A switch-expression shall not have essentially Boolean type.

Functions

17.1   (R)   The features of `<stdarg.h>` shall not be used.

17.2   (R)   Functions shall not call themselves, either directly or indirectly.

17.3   (M)   A function shall not be declared implicitly.

17.4   (M)   All exit paths from a function with non-`void` return type shall have an explicit `return` statement with an expression.

17.5   (A)   The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.

17.6   (M)   The declaration of an array parameter shall not contain the `static` keyword between the `[ ]`.

17.7   (R)   The value returned by a function having non-`void` return type shall be used.

17.8   (A)   A function parameter should not be modified.

Pointers and arrays

18.1   (R)   A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.

18.2   (R)   Subtraction between pointers shall only be applied to pointers that address elements of the same array.

18.3   (R)   he relational operators `>`, `>=`, `<` and `<=` shall not be applied to objects of pointer type except where they point into the same object.

18.4   (A)   The `+`, `-`, `+=` and `-=` operators should not be applied to an expression of pointer type.

18.5   (A)   Declarations should contain no more than two levels of pointer nesting.

18.6   (R)   The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.

18.7   (R)   Flexible array members shall not be declared.

18.8   (R)   Variable-length array types shall not be used.

Overlapping storage

19.1   (M)   An object shall not be assigned or copied to an overlapping object.

19.2   (A)   The `union` keyword should not be used.

Preprocessing directives

20.1   (A)   `#include` directives should only be preceded by preprocessor directives or comments.

20.2   (R)   The `'`, `"` or `\` characters and the `/*` or `//` character sequences shall not occur in a header file name.

20.3   (R)   The `#include` directive shall be followed by either a `<filename>` or `"filename"` sequence.

20.4 (R) A macro shall not be defined with the same name as a keyword.

20.5 (A) `#undef` should not be used.

20.6 (R) Tokens that look like a preprocessing directive shall not occur within a macro argument

20.7 (R) Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.

20.8 (R) The controlling expression of a `#if` or `#elif` preprocessing directive shall evaluate to 0 or 1.

20.9 (R) All identifiers used in the controlling expression of `#if` or `#elif` preprocessing directives shall be `#define`'d before evaluation.

20.10 (A) The `#` and `##` preprocessor operators should not be used.

20.11 (R) A macro parameter immediately following a `#` operator shall not immediately be followed by a `##` operator.

20.12 (R) A macro parameter used as an operand to the `#` or `##` operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.

20.13 (R) A line whose first token is `#` shall be a valid preprocessing directive.

20.14 (R) All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if`, `#ifdef` or `#ifndef` directive to which they are related.

Standard libraries

21.1 (R) `#define` and `#undef` shall not be used on a reserved identifier or reserved macro name.

21.2 (R) A reserved identifier or macro name shall not be declared.

21.3 (R) The memory allocation and deallocation functions of `<stdlib.h>` shall not be used.

21.4 (R) The standard header file `<setjmp.h>` shall not be used.

21.5 (R) The standard header file `<signal.h>` shall not be used.

21.6 (R) The Standard Library input/output functions shall not be used.

21.7 (R) The `atof`, `atoi`, `atol` and `atoll` functions of `<stdlib.h>` shall not be used.

21.8 (R) The library functions `abort`, `exit`, `getenv` and `system` of `<stdlib.h>` shall not be used.

21.9 (R) The library functions `bsearch` and `qsort` of `<stdlib.h>` shall not be used.

21.10 (R) The Standard Library time and date functions shall not be used

21.11 (R) The standard header file `<tgmath.h>` shall not be used.

21.12 (A) The exception handling features of `<fenv.h>` should not be used.

Resources

x 22.1 (R) All resources obtained dynamically by means of Standard Library functions shall be explicitly released.

x   22.2   (M)   A block of memory shall only be freed if it was allocated by means of a Standard Library function.

x   22.3   (R)   The same file shall not be open for read and write access at the same time on different streams.

x   22.4   (M)   There shall be no attempt to write to a stream which has been opened as read-only.

x   22.5   (M)   A pointer to a `FILE` object shall not be dereferenced.

x   22.6   (M)   The value of a pointer to a `FILE` shall not be used after the associated stream has been closed.

Chapter 21. C Implementation-defined Behavior

The TASKING VX-toolset for TriCore® C compiler fully supports the ISO-C standard, but some parts of the ISO-C standard are implementation-defined. This chapter describes how the implementation-defined areas and the locale-specific areas of the C language are implemented in the TASKING C compiler. Below are some remarks on the other behaviors as mentioned in the standard.

Unspecified behavior

Unspecified behavior is the use of an unspecified value, or other behavior where the ISO-C standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance. Some of the unspecified behaviors are relevant to users of the TASKING VX-toolset for TriCore®. Some unspecified behaviors are specified in the (E)ABI. The silicon vendor is responsible for the (E)ABI.

Undefined behavior

Undefined behavior is behavior, upon use of a non-portable or erroneous program construct or of erroneous data, for which the ISO-C standard imposes no requirements.

Some undefined behaviors may trigger a compiler error or warning. Altium does not provide any guarantees about whether or not the compiler issues an error or warning. It is important to know whether your software contains undefined behaviors since this will make the source non-portable between compiler vendors and between other processors.

The MISRA-C and CERT coding guides do not refer to undefined behaviors explicitly.

21.1. Implementation-defined Behavior

Implementation-defined behavior is unspecified behavior where each implementation documents how the choice is made.

The following sections describe the implementation-defined characteristics. The section numbers listed in parenthesis refer to the corresponding sections in the ISO-C standard. The order in this chapter is the same as used in Appendix J.3 of the ISO/IEC 9899:1999 (E) standard.

21.1.1. Translation

- How a diagnostic is identified (3.10, 5.1.1.3).

  The C compiler diagnostics are explained in Section 4.8, *C Compiler Error Messages*.

- Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2).

  White-space is retained.

21.1.2. Environment

- The mapping between physical source file multibyte characters and the source character set in translation phase 1 (5.1.1.2).

  Use of variable length encoded characters in the source file in comments and string literals is permitted. A one-on-one mapping is done without interpretation of multibyte characters.

- The name and type of the function called at program startup in a freestanding environment (5.1.2.1).

  The function called at program startup (in `cstart.c`) is called `main`. The prototype for `main` in `cstart.c` is:

  ```
  extern int main( int argc, char *argv[] );
  ```

- The effect of program termination in a freestanding environment (5.1.2.1).

  Execution is halted if the program is executed under control of a debugger, otherwise the program will loop forever in function `_Exit()`.

- An alternative manner in which the `main` function may be defined (5.1.2.2.1).

  You can change the definition of `main` by altering file `cstart.c`.

- The values given to the strings pointed to by the `argv` argument to `main` (5.1.2.2.1).

  The strings get their values from the arguments given in file `cstart.c`. The program arguments are treated case sensitive.

- What constitutes an interactive device (5.1.2.3).

  The streams `stdin`, `stdout` and `stderr` are treated as interactive devices. The debugger uses these streams with File System Simulation (FSS) windows to interact.

- The set of signals, their semantics, and their default handling (7.14).

  The signals are described in Section 13.1.19, *signal.h*.

- Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1).

  There are no other values that correspond to a computational exception. All signal values are described in Section 13.1.19, *signal.h*.

- Signals for which the equivalent of `signal(sig, SIG_IGN);` is executed at program startup (7.14.1.1).

  By default the implementation does not ignore any signals at program startup.

- The set of environment names and the method for altering the environment list used by the `getenv` function (7.20.4.5).

  There are no implementation-defined environment names that are used by the `getenv` function. A skeleton is provided for the `getenv` function in the C library, because the embedded environment has no operating system. The `getenv` function calls the name as a `void` function.

- The manner of execution of the string by the `system` function (7.20.4.6).

  A skeleton is provided for the `system()` function in the C library, because the embedded environment has no operating system. The `system()` function calls the string as a `void` function.

21.1.3. Identifiers

- Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).

  No additional multibyte characters are supported in an identifier.

- The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).

  All characters in an identifier are significant.

21.1.4. Characters

- The number of bits in a byte (3.6).

  There are eight bits in a byte.

- The values of the members of the execution character set (5.2.1).

  Only 8-bit characters are supported. The values of the execution character set are the same as that of the source character set. The same representation value is used for each member in the characters sets except for the escape sequences.

- The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).

  The following table contains an overview of the escape sequences and their byte value in the execution character set.

Escape sequence	Value
\a	7
\b	8
\f	12
\n	10
\r	13
\t	9

Escape sequence	Value
\v	11

- The value of a `char` object into which has been stored any character other than a member of the basic execution character set (6.2.5).

  Any 8-bit value can be stored in a `char` object.

- Which of `signed char` or `unsigned char` has the same range, representation, and behavior as "plain" `char` (6.2.5, 6.3.1.1).

  By default "plain" `char` is the same as specifying `signed char`. With C compiler option **--uchar** (**-u**) you can change the default to `unsigned char`.

- The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).

  The mapping is one-to-one. The values of the execution character set are the same as that of the source character set. The same representation value is used for each member in the characters sets except for the escape sequences.

- The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).

  For any character constant containing more than one character, a warning is issued and the value is truncated to type `signed char`.

- The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).

  A number of wide-character functions are available as C source code, but have not been compiled with the C library. To use complete wide-character functionality, you must recompile the libraries with the macro `WCHAR_SUPPORT_ENABLED` and keep this macro also defined when compiling your own sources.

- The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).

  By default, the "C" locale is used.

- The current locale used to convert a wide string literal into corresponding wide character codes (6.4.5).

  By default, the "C" locale is used.

- The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (6.4.5).

  All source characters can be represented in the execution character set.

21.1.5. Integers

- Any extended integer types that exist in the implementation (6.2.5).

  All types are described in Section 1.1, *Data Types*.

- Whether signed integer types are represented using sign and magnitude, two's complement, or one's complement, and whether the extraordinary value is a trap representation or an ordinary value (6.2.6.2).

  Signed integer types are represented in two's complement. The most significant bit is the sign bit. 1 is negative, 0 is positive.

- The rank of any extended integer type relative to another extended integer type with the same precision (6.3.1.1).

  All types are described in Section 1.1, *Data Types*.

- The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (6.3.1.3).

  At compile time, when converting integer types and a value does not fit in a type, the compiler issues a warning and the value is truncated. At run-time no warning or signal is given and the value is truncated.

- The results of some bitwise operations on signed integers (6.5).

  The result of E1>>E2 is E1 right shifted E2 bit positions. If E1 has a signed type and a negative value, the shift behavior is implemented as an arithmetic shift. The empty position in the most significant bit is filled with a copy of the original most significant bit.

21.1.6. Floating-Point

- The accuracy of the floating-point operations and of the library functions in <math.h> and <complex.h> that return floating-point results (5.2.4.2.2).

  A float has 24 significant mantissa bits and 8 exponent bits. A double or long double has 53 significant mantissa bits and 11 exponent bits. This is conform IEEE-754 for single precision and double precision floating-point. Internally the compiler uses a mantissa of 80 bits. The results of floating-point operations are rounded to the nearest IEEE-754 format.

  The accuracy of sqrt is defined unknown.

- The rounding behaviors characterized by non-standard values of FLT_ROUNDS (5.2.4.2.2).

  No non-standard values are used.

- The evaluation methods characterized by non-standard negative values of FLT_EVAL_METHOD (5.2.4.2.2).

  FLT_EVAL_METHOD is defined as 0. No non-standard values are used.

- The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (6.3.1.4).

  For FPU instructions, the rounding mode is used. For software floating-point instructions, the round to nearest method is used. `FLT_ROUNDS` is ignored.

- The direction of rounding when a floating-point number is converted to a narrower floating-point number (6.3.1.5).

  The round to nearest method is used. `FLT_ROUNDS` is ignored.

- How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (6.4.4.2).

  The round to nearest method is used. `FLT_ROUNDS` is ignored.

- Whether and how floating expressions are contracted when not disallowed by the `FP_CONTRACT` pragma (6.5).

  Pragma `FP_CONTRACT` is equivalent to compiler option **--fp-model=+contract**. This has only effect for fused multiply-and-accumulate (FMA) operations. FMA operations are not supported by the IEEE 754-1985 standard. The result of FMA operations is only rounded once at the end of the FMA. You can disable FMAs with the compiler option **--fp-model=-contract**.

- The default state for the `FENV_ACCESS` pragma (7.6.1).

  The default state of pragma `FENV_ACCESS` is "off". This pragma is ignored.

- Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (7.6, 7.12).

  No additional floating-point exceptions, rounding modes, environments, and classifications are defined.

- The default state for the `FP_CONTRACT` pragma (7.12.2).

  The default state of pragma `FP_CONTRACT` is set by the **contract** flag of compiler option **--fp-model**. The default state is "on".

- Whether the "inexact" floating-point exception can be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation (F.9).

  For TriCore FPU instructions the "inexact" floating-point exception is not raised. For software floating-point instructions the "inexact" floating-point exception is not supported.

- Whether the "underflow" (and "inexact") floating-point exception can be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation (F.9).

  The "underflow" floating-point exception can be raised for FPU instructions (as subnormals are not supported).

21.1.7. Arrays and Pointers

• The result of converting a pointer to an integer or vice versa (6.3.2.3).

All non-pointer conversions to and from a 32-bit pointer are implemented as a conversion to or from a 32-bit integer type.

• The size of the result of subtracting two pointers to elements of the same array (6.5.6).

The size of `ptrdiff_t` is 32 bits. The difference in address location is expressed in bytes.

21.1.8. Hints

• The extent to which suggestions made by using the `register` storage-class specifier are effective (6.7.1).

The compiler does not make assumptions based on the `register` storage-class specifier. So, basically this keyword is ignored, except that you cannot take the address of a `register` variable. The compiler issues an error in that case.

• The extent to which suggestions made by using the `inline` function specifier are effective (6.7.4).

With the `inline` keyword you force the compiler to inline the specified function, regardless of the optimization strategy of the compiler itself. For more information see Section 1.11.3, *Inlining Functions: inline*.

21.1.9. Structures, Unions, Enumerations, and Bit-fields

• Whether a "plain" `int` bit-field is treated as a `signed int` bit-field or as an `unsigned int` bit-field (6.7.2, 6.7.2.1).

By default an `int` bit-field is treated as `unsigned int`. This offers the best performance. With C compiler option **--signed-bitfields** you tell the compiler to treat `int` bit-fields as `signed int`. In this case, you can still add the keyword `unsigned` to treat a particular `int` bit-field as `unsigned`

• Allowable bit-field types other than `_Bool`, `signed int`, and `unsigned int` (6.7.2.1).

All integer types as specified in Section 1.1, *Data Types* are allowable bit-field types. To force 16-bit or 32-bit access when you declare bit-fields in special function registers, you can use the data type qualifiers `__sfrbit16` and `__sfrbit32`. See Section 1.3.2, *Accessing Hardware from C*.

• Whether a bit-field can straddle a storage-unit boundary (6.7.2.1).

A bit-field cannot straddle a storage-unit boundary. If insufficient space remains, the bit-field is put into the next unit.

• The order of allocation of bit-fields within a unit (6.7.2.1).

Allocation starts at the least significant bit up to the most significant bit. If the following bit-field fits within the same unit, it is allocated starting at the next available bit.

- The alignment of non-bit-field members of structures (6.7.2.1). This should present no problem unless binary data written by one implementation is read by another.

  The alignment of non-bit-field members of structures is the same as the alignment for data types as specified in Section 1.1, *Data Types*.

- The integer type compatible with each enumerated type (6.7.2.2).

  The compiler chooses the smallest suitable integer type (`char`, `unsigned char`, `short`, `unsigned short` or `int`), unless you use C compiler option **--integer-enumeration** (always use 32-bit integers for enumeration).

21.1.10. Qualifiers

- What constitutes an access to an object that has `volatile`-qualified type (6.7.3).

  Any reference to an object with `volatile` type results in an access. The order in which `volatile` objects are accessed is defined by the order expressed in the source code. References to non-volatile objects are scheduled in arbitrary order, within the constraints given by dependencies.

  If the compiler option **--language=+volatile** (**-Av**) is set, all references to non-volatile objects result in an access before the access to a `volatile` object that occurs subsequently in the source file takes place. The volatile access acts as a memory barrier.

21.1.11. Preprocessing Directives

- How sequences in both forms of header names are mapped to headers or external source file names (6.4.7).

  Sequences in header names are mapped to file names as is. The backslash "\" is not interpreted as an escape sequence. The backslash "\" (Windows) or forward slash "/" (Windows and UNIX) is interpreted as a standard directory separator.

- Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (6.10.1).

  A character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set.

- Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (6.10.1).

  A single-character character constant in a constant expression that controls conditional inclusion may have a negative value.

- The places that are searched for an included < > delimited header, and how the places are specified or the header is identified (6.10.2).

  How the compiler searches for include files is explained in Section 4.4, *How the Compiler Searches Include Files*.

- How the named source file is searched for in an included `" "` delimited header (6.10.2).

  How the compiler searches for include files is explained in Section 4.4, *How the Compiler Searches Include Files*.

- The method by which preprocessing tokens (possibly resulting from macro expansion) in a `#include` directive are combined into a header name (6.10.2).

  Preprocessing tokens in a `#include` directive are combined the same way as outside a `#include` directive.

- The nesting limit for `#include` processing (6.10.2).

  There is no nesting limit for `#include` processing.

- Whether the `#` operator inserts a `\` character before the `\` character that begins a universal character name in a character constant or string literal (6.10.3.2).

  The `#` operator inserts a `\` character before every `\` character in a character constant or string literal.

- The behavior on each recognized non-STDC `#pragma` directive (6.10.6).

  All non-STDC pragmas are described in Section 1.8, *Pragmas to Control the Compiler*.

- The definitions for `__DATE__` and `__TIME__` when respectively, the date and time of translation are not available (6.10.8).

  The date and time of translation are always available, macros `__DATE__` and `__TIME__` are always defined.

21.1.12. Library Functions

- Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1).

  All library functions are described in Chapter 13, *Libraries*. Some functions are not completely implemented because their implementation depends on the context where your application will run. These functions are for example all I/O related functions. Where possible, these functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment which enables you to debug your application. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior.

  In the implementation in the C library, the basic sequences of a multibyte character consist of single bytes (`MB_LEN_MAX` is set to 1). If you want full multibyte support, you need to change the C library. See the notes in the header files `stdio.h` and `wchar.h` for more information.

- The format of the diagnostic printed by the `assert` macro (7.2.1.1).

  The `assert()` function is implemented as a macro in `assert.h`. The output is:

  ```
  Assertion failed: (expression) file filename, line linenumber
  ```

when the parameter evaluates to zero.

- The representation of the floating-point status flags stored by the `fegetexceptflag` function (7.6.2.2).

  Exception flags are supported and are defined in header file `fenv.h`. The floating-point exception bits correspond to the bits in the highest word of PSW:

  ```
  #define FE_DIVBYZERO  (1 << 4)
  #define FE_INEXACT    (1 << 2)
  #define FE_INVALID    (1 << 6)
  #define FE_OVERFLOW   (1 << 5)
  #define FE_UNDERFLOW  (1 << 3)
  ```

- Whether the `feraiseexcept` function raises the "inexact" floating-point exception in addition to the "overflow" or "underflow" floating-point exception (7.6.2.3).

  Function `feraiseexcept` does not raise `FE_INEXACT` if underflow or overflow is raised.

- Strings other than `"C"` and `""` that may be passed as the second argument to the `setlocale` function (7.11.1.1).

  No other strings are predefined. A NULL pointer as the second argument returns the "C" locale. Any other string than `"C"` or `""` can be passed as the second argument to the `setlocale` function and results in NULL.

- The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 or greater than 2 (7.12).

  The `FLT_EVAL_METHOD` macro can only have the values 0, 1 or 2.

- Domain errors for the mathematical functions, other than those required by this International Standard (7.12.1).

  No other domain errors exist, other than those required by the standard.

- The values returned by the mathematical functions on domain errors (7.12.1).

  On domain errors (`errno` is set to `EDOM`), the mathematical functions return a value as specified in the following table.

Math function	Return value on EDOM
`acos(|x| > 1.0)`	0.0
`asin(|x| > 1.0)`	0.0
`log(x < 0.0)`	`-HUGE_VAL`
`pow(x <= 0.0)`	0.0
`sqrt(x < 0.0)`	-NaN

- The values returned by the mathematical functions on underflow range errors, whether `errno` is set to the value of the macro `ERANGE` when the integer expression `math_errhandling & MATH_ERRNO` is nonzero, and whether the "underflow" floating-point exception is raised when the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero. (7.12.1).

  On underflow range errors, the mathematical functions return 0.0. `math_errhandling` is set to `MATH_ERRNO`. Trapping and non-trapping versions of the library are available. With a non-trapping library `errno` is not set to `ERANGE` on underflow range errors, with a trapping library no underflow exception is raised.

- Whether a domain error occurs or zero is returned when an `fmod` function has a second argument of zero (7.12.10.1).

  Zero (`0.0`) is returned when an `fmod` function has a second argument of zero.

- The base-2 logarithm of the modulus used by the `remquo` functions in reducing the quotient (7.12.10.3).

  The `remquo` function calculates at least 8 bits of the quotient.

- Whether the equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1).

  The equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler.

- The null pointer constant to which the macro `NULL` expands (7.17).

  Macro `NULL` is defined as `(void *) 0`.

- Whether the last line of a text stream requires a terminating new-line character (7.19.2).

  Both a new-line character (`\n`) and end-of-file (`EOF`) are recognized as the termination character of a line.

- Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.19.2).

  Space characters written to a stream immediately before a new-line character are preserved.

- The number of null characters that may be appended to data written to a binary stream (7.19.2).

  I/O related functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior. The library does not append any null characters. It depends on the `open()` function on the host environment what happens. You can write your own `_open()` function if necessary.

- Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.19.3).

  I/O related functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior. Where the

file position indicator of an append-mode stream is initially positioned depends on the `open()` function on the host environment. You can write your own `_open()` function if necessary.

- Whether a write on a text stream causes the associated file to be truncated beyond that point (7.19.3).

  I/O related functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior. Whether a write on a text stream causes the associated file to be truncated beyond that point depends on how the low-level file routines are implemented in your application.

- The characteristics of file buffering (7.19.3).

  Files can be unbuffered, fully buffered or line buffered. What actually happens depends on how the low-level file routines are implemented in your application.

- Whether a zero-length file actually exists (7.19.3).

  This depends on how the low-level file routines are implemented in your application.

- The rules for composing valid file names (7.19.3).

  This depends on how the low-level file routines are implemented in your application.

- Whether the same file can be simultaneously open multiple times (7.19.3).

  This depends on how the low-level file routines are implemented in your application.

- The nature and choice of encodings used for multibyte characters in files (7.19.3).

  Use of variable length encoded characters in files in comments and string literals is permitted.

- The effect of the `remove` function on an open file (7.19.4.1).

  This depends on how the low-level file routines are implemented in your application.

- The effect if a file with the new name exists prior to a call to the `rename` function (7.19.4.2).

  This depends on how the low-level file routines are implemented in your application.

- Whether an open temporary file is removed upon abnormal program termination (7.19.4.3).

  This depends on how the low-level file routines are implemented in your application.

- Which changes of mode are permitted (if any), and under what circumstances (7.19.5.4).

  The `freopen()` function first calls `fclose()` and then calls `_fopen()` with the new mode.

- The style used to print an infinity or NaN, and the meaning of any n-char or n-wchar sequence printed for a NaN (7.19.6.1, 7.24.2.1).

  The style used to print an infinity or NaN is `inf` and `nan` respectively (`INF` or `NAN` for the `F` conversion specifier). n-char or w-char sequences are not used for `nan`.

- The output for `%p` conversion in the `fprintf` or `fwprintf` function (7.19.6.1, 7.24.2.1).

  The argument is treated as having type `void *`. The value will be printed as a hexadecimal value, similar to `%x`.

- The interpretation of a – character that is neither the first nor the last character, nor the second where a ^ character is the first, in the scanlist for `%[` conversion in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.1).

  A – character is treated as a normal character.

- The set of sequences matched by a `%p` conversion and the interpretation of the corresponding input item in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.2).

  The format of `%p` matches the format of `%x`. The input for `%p` is a hexadecimal value, which is converted to a value with type `void *`.

- The value to which the macro `errno` is set by the `fgetpos`, `fsetpos`, or `ftell` functions on failure (7.19.9.1, 7.19.9.3, 7.19.9.4).

  If `errno` is set to a value depends on how the low-level file routines are implemented in your application.

- The meaning of any n-char or n-wchar sequence in a string representing a NaN that is converted by the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function (7.20.1.3, 7.24.4.1.1).

  An n-char or n-wchar sequence in a string representing a NaN is ignored.

- Whether or not the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function sets `errno` to `ERANGE` when underflow occurs (7.20.1.3, 7.24.4.1.1).

  `errno` is set to `ERANGE` when underflow occurs and the value returned is 0.0.

- Whether the `calloc`, `malloc`, and `realloc` functions return a null pointer or a pointer to an allocated object when the size requested is zero (7.20.3).

  `NULL` is returned when a size of zero is requested.

- Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed when the `abort` or `_Exit` function is called (7.20.4.1, 7.20.4.4).

  When the `abort()` or `_Exit()` function is called, open streams with unwritten buffered data are not flushed, open streams are not closed, and temporary files are not removed.

- The termination status returned to the host environment by the `abort`, `exit`, or `_Exit` function (7.20.4.1, 7.20.4.3, 7.20.4.4).

  `exit()` and `_Exit()` use the input value as termination status. `abort()` calls `_Exit()` with `EXIT_FAILURE`.

- The value returned by the `system` function when its argument is not a null pointer (7.20.4.6).

  A skeleton is provided for the `system()` function in the C library, because the embedded environment has no operating system. The `system()` function returns the value 0.

- The local time zone and Daylight Saving Time (7.23.1).

  The default time zone is UTC. Daylight Saving Time is not available (`tm_isdst=-1`).

- The range and precision of times representable in `clock_t` and `time_t` (7.23).

  `clock_t` is defined as `unsigned long long`, `time_t` is defined as `unsigned long`. The resolution of the clock is defined by `CLOCKS_PER_SEC`, which is set by means of the C startup code (see the `setclockpersec()` function in `cstart.c`).

- The era for the `clock` function (7.23.2.1).

  The `clock` function returns the current processor time.

- The replacement string for the `%Z` specifier to the `strftime`, and `wcsftime` functions in the `"C"` locale (7.23.3.5, 7.24.5.1).

  `%Z` is replaced by the time zone name, by default UTC.

- Whether or when the trigonometric, hyperbolic, base-e exponential, base-e logarithmic, error, and log gamma functions raise the "inexact" floating-point exception in an IEC 60559 conformant implementation (F.9).

  The "inexact" floating-point exception is not supported.

- Whether the functions in `<math.h>` honor the rounding direction mode in an IEC 60559 conformant implementation (F.9).

  The round to nearest method is used. `FLT_ROUNDS` is defined as 1.

21.1.13. Architecture

- The values or expressions assigned to the macros specified in the headers `<float.h>`, `<limits.h>`, and `<stdint.h>` (5.2.4.2, 7.18.2, 7.18.3).

Macros in `<float.h>`:

Macro <float.h>	Value
FLT_RADIX	2
FLT_ROUNDS	1 (round to nearest)
FLT_EVAL_METHOD	0
FLT_MANT_DIG	24
FLT_DIG	6
FLT_EPSILON	1.19209290E-07F
FLT_MIN	1.17549435E-38F
FLT_MIN_EXP	-125
FLT_MIN_10_EXP	-37

Macro <float.h>	Value
`FLT_MAX`	3.40282347E+38F
`FLT_MAX_EXP`	+128
`FLT_MAX_10_EXP`	+38
`[L]DBL_MANT_DIG`	53
`[L]DBL_DIG`	15
`[L]DBL_EPSILON`	2.2204460492503131E-16
`[L]DBL_MIN`	2.2250738585072014E-308
`[L]DBL_MIN_EXP`	-1021
`[L]DBL_MIN_10_EXP`	-307
`[L]DBL_MAX`	1.7976931348623157E+308
`[L]DBL_MAX_EXP`	+1024
`[L]DBL_MAX_10_EXP`	+308
`DECIMAL_DIG`	17 (for double FP), 9 (for single FP)
`FLT16_MANT_DIG`	11
`FLT16_DIG`	3
`FLT16_EPSILON`	9.765625E-4F
`FLT16_MIN`	6.103515625E-05F
`FLT16_MIN_EXP`	-13
`FLT16_MIN_10_EXP`	-4
`FLT16_MAX`	65504.0F
`FLT16_MAX_EXP`	+16
`FLT16_MAX_10_EXP`	+4
`FLT16_HAS_SUBNORM`	1
`FLT16_TRUE_MIN`	_FLT16_MIN
`FLT16_DECIMAL_DIG`	5
`SSACCUM_INT_DIG`	17
`SACCUM_INT_DIG`	17
`ACCUM_INT_DIG`	17
`LACCUM_INT_DIG`	17
`SSFRACT_FRACT_DIG`	15
`SFRACT_FRACT_DIG`	15
`FRACT_FRACT_DIG`	31
`LFRACT_FRACT_DIG`	31
`SSACCUM_FRACT_DIG`	46

Macro <float.h>	Value
SACCUM_FRACT_DIG	46
ACCUM_FRACT_DIG	46
LACCUM_FRACT_DIG	46
SSFRACT_MAX	(__ssfract) 0x0.fffep0hr
SFRACT_MAX	(__sfract) 0x0.fffep0hr
FRACT_MAX	(__fract) 0x0.fffffffep0r
LFRACT_MAX	(__lfract) 0x0.fffffffep0lr
LLFRACT_MAX	(__lfract) 0x0.fffffffep0lr
SSACCUM_MAX	(__ssaccum) 0x1ffff.fffffffffffcp0hk
SACCUM_MAX	(__saccum) 0x1ffff.fffffffffffcp0hk
ACCUM_MAX	(__accum) 0x1ffff.fffffffffffcp0k
LACCUM_MAX	(__laccum) 0x1ffff.fffffffffffcp0lk
LLACCUM_MAX	(__laccum) 0x1ffff.fffffffffffcp0lk
SSFRACT_MIN	(__ssfract) -1.0
SFRACT_MIN	(__sfract) -1.0
FRACT_MIN	(__fract) -1.0
LFRACT_MIN	(__lfract) -1.0
LLFRACT_MIN	(__lfract) -1.0
SSACCUM_MIN	(__ssaccum) -131072.0
SACCUM_MIN	(__saccum) -131072.0
ACCUM_MIN	(__accum) -131072.0
LACCUM_MIN	(__laccum) -131072.0
LLACCUM_MIN	(__laccum) -131072.0
SSFRACT_EPSILON	(__ssfract) 0x0.0002p0hr
SFRACT_EPSILON	(__sfract) 0x0.0002p0hr
FRACT_EPSILON	(__fract) 0x0.00000002p0r
LFRACT_EPSILON	(__lfract) 0x0.00000002p0lr
LLFRACT_EPSILON	(__lfract) 0x0.00000002p0lr
SSACCUM_EPSILON	(__ssaccum) 0x0.000000000004p0hk
SACCUM_EPSILON	(__saccum) 0x0.000000000004p0hk
ACCUM_EPSILON	(__accum) 0x0.000000000004p0k
LACCUM_EPSILON	(__laccum) 0x0.000000000004p0lk
LLACCUM_EPSILON	(__laccum) 0x0.000000000004p0lk

Macros in `<limits.h>`:

Macro `<limits.h>`	Value
`CHAR_BIT`	8
`SCHAR_MIN`	`-SCHAR_MAX-1`
`SCHAR_MAX`	0x7f
`UCHAR_MAX`	0xffU
`CHAR_MIN`	`__CHAR_MIN` (min value of 'plain' char)
`CHAR_MAX`	`__CHAR_MAX` (max value of 'plain' char)
`MB_LEN_MAX`	1
`SHRT_MIN`	`-SHRT_MAX-1`
`SHRT_MAX`	0x7fff
`USHRT_MAX`	0xffffU
`INT_MIN`	`-INT_MAX-1`
`INT_MAX`	0x7fffffff
`UINT_MAX`	0xffffffffU
`LONG_MIN`	`-LONG_MAX-1`
`LONG_MAX`	0x7fffffffL
`ULONG_MAX`	0xffffffffUL
`LLONG_MIN`	`-LLONG_MAX-1`
`LLONG_MAX`	0x7fffffffffffffffLL
`ULLONG_MAX`	0xffffffffffffffffULL

The limit macros in `<stdint.h>` for exact-width, minimum-width and fastest-width integer types have the same ranges as `char`, `short`, `int`, `long` and `long long`. Furthermore the following macros are defined:

Macro `<stdint.h>`	Value
`INTPTR_MIN`	`INT32_MIN`
`INTPTR_MAX`	`INT32_MAX`
`UINTPTR_MAX`	`UINT32_MAX`
`INTMAX_MIN`	`INT64_MIN`
`INTMAX_MAX`	`INT64_MAX`
`UINTMAX_MAX`	`UINT64_MAX`
`PTRDIFF_MIN`	`__PTRDIFF_MIN`
`PTRDIFF_MAX`	`__PTRDIFF_MAX`
`SIG_ATOMIC_MIN`	`INT32_MIN`
`SIG_ATOMIC_MAX`	`INT32_MAX`

Macro <stdint.h>	Value
SIZE_MAX	__SIZE_MAX
WCHAR_MIN	__WCHAR_MIN
WCHAR_MAX	__WCHAR_MAX
WINT_MIN	0
WINT_MAX	UINT32_MAX

- The number, order, and encoding of bytes in any object (when not explicitly specified in this International Standard) (6.2.6.1).

  All types are described in Section 1.1, *Data Types*.

- The value of the result of the sizeof operator (6.5.3.4).

  The value of the size of the data types is described in Section 1.1, *Data Types*. Divide the size by 8 because the table lists the size of the data types in bits.

21.2. Locale-specific Behavior

Locale-specific behavior is behavior that depends on local conventions of nationality, culture, and language that each implementation documents.

The following items describe the locale-specific characteristics, as indicated in Appendix J.4 of the ISO/IEC 9899:1999 (E) standard.

- Additional members of the source and execution character sets beyond the basic character set (5.2.1).

  The compiler accepts all one-byte characters in the host's default character set. Use of variable length encoded characters in the source file in comments and string literals is permitted.

  In the implementation in the C library, the basic sequences of a multibyte character consist of single bytes (MB_LEN_MAX is set to 1). If you want full multibyte support, you need to change the C library. See the notes in the header files stdio.h and wchar.h for more information.

- The presence, meaning, and representation of additional multibyte characters in the execution character set beyond the basic character set (5.2.1.2).

  Use of variable length encoded characters in the source file in comments and string literals is permitted.

- The shift states used for the encoding of multibyte characters (5.2.1.2).

  A multibyte character must be a single byte when in the initial shift state.

- The direction of writing of successive printing characters (5.2.2).

  The direction of writing depends on the application and the display device.

- The decimal-point character (7.1.1).

  The default decimal-point character is a `'.'`.

- The set of printing characters (7.4, 7.25.2).

  The set of printing characters are the characters for which the `isprint()` function returns true. Printing characters are characters in the range 32 (space) to 126.

- The set of control characters (7.4, 7.25.2).

  The set of control characters are the characters for which the `iscntrl()` function returns true. Control characters are characters in the range 0 to 31 and 127.

- The sets of characters tested for by the `isalpha`, `isblank`, `islower`, `ispunct`, `isspace`, `isupper`, `iswalpha`, `iswblank`, `iswlower`, `iswpunct`, `iswspace`, or `iswupper` functions (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.2.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11).

  The characters tested for are specified in the following table.

Function	Characters tested
`isalpha`	a-z, A-Z
`isblank`	`'  '` (space), `'\t'` (tab)
`islower`	a-z
`ispunct`	!, ", #, $, %, &, ', (,), *, +, ,, -, ., /, :, ;, <, =, >, ?, @, [, \,], ^, _, `, {, |, }, ~
`isspace`	`'  '` (space), `'\t'`, `'\n'`, `'\v'`, `'\f'`, `'\r'`
`isupper`	A-Z

- The native environment (7.11.1.1).

  The native environment is the same as the "C" locale.

- Additional subject sequences accepted by the numeric conversion functions (7.20.1, 7.24.4.1).

  No additional subject sequences are accepted.

- The collation sequence of the execution character set (7.21.4.3, 7.24.4.4.2).

  Only the "C" locale is supported. The `strcoll()` function is the same as the `strcmp()` function. The `wcscoll()` function is the same as the `wcscmp()` function.

- The contents of the error message strings set up by the `strerror` function (7.21.6.2).

  The error message strings returned by `strerror()` depend on the argument. Typically, the values for the argument come from `errno.h`. For a list of messages see Section 13.1.6, *errno.h*).

- The formats for time and date (7.23.3.5, 7.24.5.1).

  English names for months and days are used.

`%c` is replaced by the following date and time representation: `%a %b %e %H:%M:%S %Y`

`%x` is replaced by the following date representation: `%m/%d/%y`

`%X` is replaced by the following time representation: `%H:%M:%S`

- Character mappings that are supported by the `towctrans` function (7.25.1).

  The character mappings supported by the `towctrans()` function are defined in `wctype.h`: `_to_lower` and `_to_upper`.

- Character classifications that are supported by the `iswctype` function (7.25.1).

  The character classifications supported by the `iswctype()` function are defined in `wctype.h`: `_alnum`, `_alpha`, `_cntrl`, `_digit`, `_graph`, `_lower`, `_print`, `_punct`, `_space`, `_upper`, `_xdigit` and `_blank`.