TASKING VX-toolset for 8051 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 C compiler for 8051 fully supports the ISO-C standard and add extra possibilities to program the special functions of the target.

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

* keywords to specify memory types for data and functions

* attribute to specify 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

Fundamental Data Types

The C compiler supports the ISO C99 defined data types. The sizes of these types are shown in the following table.

C Type	Size	Align	Limits
__bit	1	1	0 or 1
_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	8	$[-2^{15}, 2^{15}-1]$
unsigned short	16	8	$[0, 2^{16}-1]$
int	16	8	$[-2^{15}, 2^{15}-1]$

C Type	Size	Align	Limits
unsigned int	16	8	$[0, 2^{16}\text{-}1]$
enum [*]	1 8 16	1 8 8	0 or 1 $[-2^7, 2^7\text{-}1]$ or $[0, 2^8\text{-}1]$ $[-2^{15}, 2^{15}\text{-}1]$ or $[0, 2^{16}\text{-}1]$
long	32	8	$[-2^{31}, 2^{31}\text{-}1]$
unsigned long	32	8	$[0, 2^{32}\text{-}1]$
long long [**]	32	8	$[-2^{31}, 2^{31}\text{-}1]$
unsigned long long [**]	32	8	$[0, 2^{32}\text{-}1]$
float (23-bit mantissa)	32	8	[–3.402E+38, –1.175E-38] [+1.175E-38, +3.402E+38]
double [**] long double	32	8	[–3.402E+38, –1.175E-38] [+1.175E-38, +3.402E+38]
pointer to __sfr, __bsfr, __data, __bdata, __idata, __pdata or __bit	8	8	$[0, 2^8\text{-}1]$
pointer to function, __xdata or __rom	16	8	$[0, 2^{16}\text{-}1]$

[*] When you use the `enum` type, the compiler will use the smallest sufficient type (`__bit`, `char`, `unsigned char` or `int`), unless you use C compiler option **--integer-enumeration** (always use 16-bit integers for enumeration).

[**] The `long long` types are treated as `long`. The `double` and `long double` types are always treated as `float`.

Bit Data Type

You can use the `__bit` type to define scalars in the bit-addressable area and for the return type of functions. A struct containing bit-fields cannot be used for this purpose, for example because the struct is aligned at a byte boundary. Unlike the `_Bool` type the `__bit` type is aligned on a bit boundary.

The following rules apply to `__bit` type variables:

• A `__bit` type variable is always unsigned.

• A `__bit` type variable can be exchanged with all other type-variables. The compiler generates the correct conversion.

A `__bit` type variable is like a boolean. Therefore, if you convert an `int` type variable to a `__bit` type variable, it becomes 1 (true) if the integer is not equal to 0, and 0 (false) if the integer is 0. The next two C source lines have the same effect:

```
bit_variable = int_variable;
bit_variable = int_variable ? 1 : 0;
```

- Pointer to `__bit` is allowed, but you cannot take the address of a bit on the stack.

- The `__bit` type is allowed as a structure member. However, a bit structure can only contain members of type `__bit`, and you cannot push a bit structure on the stack or return a bit structure via a function.

- A union of a `__bit` structure and another type is not allowed.

- A `__bit` type variable is allowed as a parameter of a function.

- A `__bit` type variable is allowed as a return type of a function.

- A `__bit` typed expression is allowed as switch expression.

- The `sizeof` of a `__bit` type is 1.

- A global or static `__bit` type variable can be initialized.

- A `__bit` type variable can be declared volatile.

__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 )
```

1.2. Accessing Memory

The TASKING VX-toolset for 8051 toolset has several keywords you can use in your C source to specify memory locations. This is explained in the sub-sections that follow.

1.2.1. Memory Type 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 type qualifier*. If you do not specify a memory type qualifier, data objects get a default memory type based on the memory model.

You can specify the following memory types:

Qualifier	Description	Location	Maximum object size	Pointer size	Pointer arithmetic	Section type[*]
__bdata	Bit addressable	Bit addressable memory in internal RAM	Size of bit addressable memory	8-bit	8-bit	bdata
__data	Direct addressable internal RAM data	Lower 128 bytes in internal RAM	128 bytes	8-bit	8-bit	data
__idata	Indirect addressable internal RAM data	Internal RAM	Size of internal RAM	8-bit	8-bit	idata

Qualifier	Description	Location	Maximum object size	Pointer size	Pointer arithmetic	Section type[*]
__sfr	Special function register	Upper 128 bytes in internal RAM	No allocation possible[**]	8-bit	8-bit	--
__bsfr	Bit addressable special function register	Upper 128 bytes in internal RAM	No allocation possible[**]	8-bit	8-bit	--
__xdata	External RAM data	External RAM	64 kB	16-bit	16-bit	xdata
__pdata	Page in external RAM data	External RAM	256 bytes	8-bit	8-bit	pdata
__rom	External ROM data	External ROM	64 kB	16-bit	16-bit	rom

[*] The default section name is equal to the section type followed by a single underscore and the name of the allocated object. You can change the section name with the #pragma section or command line option **--rename-sections**.

[**] Because the SFR area has a predefined layout (little-endian), it is not possible to allocate variables in this area. The SFR area is only accessible through a direct addressing mode. Therefore, a warning will be generated when a pointer to __sfr or __bsfr is dereferenced.

Examples using explicit memory types

```
__data   char  c;
__rom    char  text[] = "No smoking";
__xdata  int   array[10][4];
__idata  long  l;
```

The memory type qualifiers are treated like any other data type specifier (such as unsigned). This means the examples above can also be declared as:

```
char __data   c;
char __rom    text[] = "No smoking";
int  __xdata  array[10][4];
long __idata  l;
```

1.2.1.1. Pointers with Memory Type Qualifiers

Pointers for the 8051 can have two types: a 'logical' type and a memory type. For example,

```
__rom char *__data p;  /* pointer residing in data, pointing to ROM */
```

means p has memory type __data (p itself is allocated in on-chip RAM), but has logical type 'character in target memory space ROM'. The memory type qualifier used to the left of the '*', specifies the target memory of the pointer, the memory type qualifier used to the right of the '*', specifies the storage memory of the pointer.

The 8051 C compiler is very efficient in allocating pointers, because it recognizes far (2 byte) and near (1 byte) pointers. Pointers to __data, __idata, __pdata, __bdata and __bit have a size of 1 byte, whereas pointers to __rom, __xdata and functions (in ROM) have a size of 2 bytes.

Pointer conversions

Conversions of pointers with the same qualifiers are always allowed. The following table contains the additionally allowed pointer conversions. Other pointer conversions are not allowed to avoid possible run-time errors.

Source pointer	Destination pointer
__bdata	__data
__bdata	__idata
__data	__idata
__pdata	__xdata
__bsfr	__sfr

1.2.1.2. Structure Tags with Memory Type Qualifiers

A tag declaration is intended to specify the layout of a structure or union. If a memory type is specified, it is considered to be part of the declarator. The tag name itself, nor its members can be bound to any storage area, although members having type "... pointer to" do require one. The tag may then be used to declare objects of that type, and may allocate them in different memories. The following example illustrates this constraint.

```
struct S {
  __xdata int i;   /* referring to storage: not correct      */
  __idata char *p; /* used to specify target memory: correct */
};
```

In the example above the 8051 compiler ignores the erroneous __xdata memory type qualifier (and issues a warning message).

1.2.1.3. Typedefs with Memory Type Qualifiers

Typedef declarations follow the same scope rules as any declared object. Typedef names may be (re-)declared in inner blocks but not at the parameter level. However, in typedef declarations, memory type qualifiers are allowed. A typedef declaration should at least contain one type qualifier.

Example using memory types with typedefs:

```
typedef __idata int IDATINT;   /* memory type __idata: OK */
typedef int __data *DATAPTR;   /* logical type __data,
                                  memory type 'default'   */
```

1.2.2. Memory Models

The C compiler supports three data memory models, listed in the following table.

Memory model	Description	Letter	Max RAM size	Default data memory type
Small	Direct addressable internal RAM	s	128 bytes	`__data`
Auxiliary page	One page of external RAM	a	256 bytes	`__pdata`
Large	External RAM	l	64 kB	`__xdata`

Each memory model defines a default memory type for objects that do not have a memory type qualifier specified. By default, the 8051 compiler uses the small memory model. With the C compiler option **--model** you can specify another memory model. Per memory model you can choose to use reentrancy which enables you to call functions recursively.

You can overrule the default memory type with one of the memory type qualifiers. This allows you to exceed the default maximum RAM size. For information on the memory types, see Section 1.2.1, *Memory Type Qualifiers*.

Small memory model

By default the 8051 compiler uses the small memory model. In the small memory model all data objects with the default memory type and the stack (used for function parameter passing) must fit in the direct addressable area of internal RAM. Objects with an explicit memory type qualifier can exceed this limitation (for example an object qualified as `__xdata` or `__pdata`). Note that the stack length depends upon the nesting depth of the various functions. Accessing data in internal RAM is considerably faster than accessing data in external RAM. Therefore, it is useful to place often used variables in internal data memory and less often referenced data elements in external data memory.

Large memory model

When the compiler uses the large memory model to access data, the produced code is larger and in some cases slower than the code for a similar operation in one of the other memory models.

Auxiliary page memory model

The auxiliary page memory model is especially interesting for derivatives with 256 bytes of 'external' RAM on chip. All data objects with the default memory type must fit in one 256 bytes page.

Reentrancy

Optionally you can choose to enable reentrancy. If you select reentrancy, a (less efficient) virtual dynamic stack is used which allows you to call functions recursively. With reentrancy, you can call functions at any time, even from interrupt functions.

Select the memory model in Eclipse

To select the memory model to compile for:

1. Select **C Compiler » Memory Model**.

2. Select the **Small**, **Auxiliary** or **Large** compiler memory model.

3. Optionally enable the option **Allow reentrant functions**.

__MODEL__

The compiler defines the preprocessor symbol __MODEL__ to the letter representing the selected memory model. This can be very helpful in making conditional C code in one source module, used for different applications in different memory models.

Example:

```
#if __MODEL__ == 's'
/* this part is only for the small memory model */
...
#endif
```

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

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

With the attribute __at() you can specify an absolute 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;
```

The variable i is placed at address 0x1000 and is initialized.

```
void f(void) __at( 0xf0ff + 1 ) { }
```

The function f is placed at address 0xf100.

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. Otherwise the compiler generates an error.

• You can place only global variables at absolute addresses. Parameters of functions, or automatic variables within functions cannot be placed at absolute addresses. If they are, the compiler generates an error.

• A variable that is declared extern, is not allocated by the compiler in the current module. Hence you should not use the keyword __at() on an external variable. If you do, the compiler ignores the keyword __at() without generating an error. Use __at() at the definition of the variable.

- You cannot place structure members at an absolute address. If you do, the compiler ignores the keyword `__at()` and generates a warning.

- 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.

1.2.4. Accessing Bits

There are several methods to access single bits in the bit-addressable area. 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.

```
__bdata 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 shift/and/or combinations to perform the required result.

```
__bdata unsigned 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 __bdata 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;
}
```

Declaring a bit variable with __atbit() (backwards compatibility only)

For backwards compatibility, you can still use the `__atbit()` keyword to define a bit symbol as an alias for a single bit in a bit-addressable object. However, we recommend that you use one of the methods described above to access a bit.

The syntax of `__atbit()` is:

```
__atbit(object,offset)
```

where, *object* is a bit-addressable object and *offset* is the bit position in the object.

The following restrictions apply:

• This keyword can only be applied to `__bit` type symbols.

• The bit must be defined `volatile` explicitly. The compiler issues an error if the bit is not defined volatile.

- The bitword can be any `volatile` bit-addressable (`__bdata`) object. The compiler issues an error if the bit-addressable object was not volatile.

- The bit symbol cannot be used as a global symbol. An extern on the bit variable, without `__atbit()`, will lead to an unresolved external message from the linker, so therefore `__atbit()` is required.

Examples

```
/* Module 1 */
volatile __bdata unsigned int bitword;
volatile __bit b __atbit( bitword, 3 );

/* Module 2 */
extern volatile __bdata unsigned int bitword;
extern volatile __bit b __atbit( bitword, 3);
```

Drawbacks of __atbit()

The `__atbit()` requires all involved objects to be volatile. If your application does not require these objects to be volatile, you may see in many cases that the generated code is less optimal than when the objects were not volatile. The reason for that is that the compiler must generate each read and write access for volatile objects as written down in the C code. Fortunately the standard C language provides methods to achieve the same result as with `__atbit()`. The compiler is smart enough to generate efficient bit operations where possible.

1.2.5. Accessing Hardware from C: __sfr, __bsfr

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. These SFR files are also used by the assembler and the simulator engine. The debugger and the Eclipse IDE use the XML variants of the SFR files. The XML files include full descriptions of the SFRs and the bit-fields. Also the bit-field values are described. To decrease compile time the `.sfr` files do not contain the descriptions. The `.sfr` files are in written C and are derived from the XML files.

Example use in C:

```
#include <regtc26x.sfr>      // include the SFR file

void set_sfr(void)
{
    SCON = 0x88;             // use SFR name
    SCR_P00_IN_3 = 1;        // use of bit name
    if (SCR_P00_IN_4 == 1)
    {
        SCR_P00_IN_3 = 0;
    }
    TCON_IE1 = 1;            // use of bit name
}
```

You can find a list of defined SFRs and defined bits by inspecting the SFR file for a specific processor. The files are named reg*cpu*.sfr, where *cpu* is the name of the target processor. You can include the register file you want use in your source manually or you can specify control program option **--include-sfr-file**. The files are located in the standard include directory.

Defining Special Function Registers

With the __sfr memory type qualifier you can define a symbol as an SFR. The compiler may assume that special SFR operations can be performed on such symbols. The compiler can decide to use bit instructions for those special function registers that are bit accessible, in this case use __bsfr instead of __sfr. For example, if bits are defined in the SFR definition, these bits can be accessed using bit instructions.

Note that the __sfr space is little-endian, while the other spaces are big-endian.

For the 8051 only the SFRs at addresses 0x80, 0x88, 0x90, 0x98, 0xa0, 0xa8, 0xb0, 0xb8, 0xc0, 0xc8, 0xd0, 0xd8, 0xe0, 0xe8, 0xf0 and 0xf8 are bit addressable.

A typical definition of a special function register looks as follows:

```
typedef struct
    _Bool  __b0 : 1;
    _Bool  __b1 : 1;
    _Bool  __b2 : 1;
    _Bool  __b3 : 1;
    _Bool  __b4 : 1;
    _Bool  __b5 : 1;
    _Bool  __b6 : 1;
    _Bool  __b7 : 1;
} __bitstruct_t;

#define SP        (*(__sfr volatile unsigned char *) 0x81)

#define TCON      (*(__bsfr volatile unsigned char *) 0x88)
#define TCON_IT0  ((*(__bsfr volatile __bitstruct_t *) 0x88).__b0)
#define TCON_IE0  ((*(__bsfr volatile __bitstruct_t *) 0x88).__b1)
```

Because the special function registers are dealing with I/O, they are declared volatile. It is incorrect to optimize away the access to them.

1.3. 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];
}
```

1.4. 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. 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*
%*parm_nr*	Parameter number in the range 0 .. 9.
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
r	register	R0 - R7	input/output constraint To be used in places where an Rn addressing mode is allowed. It can be turned into a direct addressing mode by using an explicit "A" prefix in the inline assembly code.
b	bit register	B.0 - B.7, F0, F1	input/output constraint
s	indirect address register	R0 - R1	input constraint only

Constraint character	Type	Operand	Remark
d	direct address register	AR0 - AR7, B and the pseudo registers	input/output constraint To be used in places where a direct addressing mode is allowed.
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 7). 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. A register is chosen for the parameter because of the constraint r; 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 assign the result to the output variable.

```
__data char out;
void get_out( void )
{
    __asm( "mov %0,#0xff"
           : "=r" (out) );
}
```

Generated assembly code:

```
mov R0,#0xff
mov    _out,R0
```

Example 3: using input parameters

Assign a variable to an SFR. A register is chosen for the parameter because of the constraint `r`; 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.

```
__data char in;
void init_sfr( void )
{
    __asm( "MOV  P0,%0"
            :
            : "r" (in) );
}
```

Generated assembly code:

```
    mov    R0,_in
    MOV  P0,R0
```

Example 4: using input and output parameters

Add two C variables and assign the result to a third C variable. Registers are necessary for the input and output parameters (constraint `r`, `%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 variables.

```
__data char in1, in2, out;
void add2( void )
{
    __asm( "MOV A, %1\n\t"
            "ADD A, %2\n\t"
            "MOV %0, A"
            : "=r" (out)
            : "r" (in1), "r" (in2) );
}

void main(void)
{
    in1 = 3;
    in2 = 4;
    add2( );
}
```

Generated assembly code:

```
_add2:
; Code generated by C compiler
    mov    R0,_in2
    mov    R1,_in1
; __asm statement expansion
    MOV A, R1
    ADD A, R0
    MOV R0, A
; Code generated by C compiler
    mov    _out, R0
_main:
    mov    _in1,#3
    mov    _in2,#4
```

Example 5: using an explicit "A" prefix to turn a "r" constraint into a direct addressing mode

```
__data char in, out;
void m_inc( void )
{
    __asm( "MOV %0,A%1\n\t"
           "INC %0"
           : "=r" (out)
           : "r" (in) );
}
```

Generated assembly code:

```
_m_inc:
; Code generated by C compiler
    mov    R0,_in
; __asm statement expansion
    MOV R0,AR0
    INC R0
; Code generated by C compiler
    mov    _out,R0
```

When you use the "d" constraint a pseudo-register might also be used. GPRs will be prefixed with an "A" automatically:

```
__data char in, out;
void m_inc2( void )
{
    __asm( "MOV %0,%1\n\t"
           "INC %0"
           : "=r"(out)
           : "d"(in) );
}
```

Generated assembly code is the same.

Example 6: 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 R0 is a reserved register. You can do this by adding a reserved register list (: "R0"). As you can see in the generated assembly code, register R0 is not used (the first register used is R1).

```
__data char in1, in2, out;
void add2( void )
{
    __asm( "MOV A, %1\n\t"
           "ADD A, %2\n\t"
           "MOV %0, A"
           : "=r" (out)
           : "r" (in1), "r" (in2)
           : "R0" );
}

void main(void)
{
    in1 = 3;
    in2 = 4;
    add2( );
}
```

Generated assembly code:

```
_add2:
; Code generated by C compiler
    mov   R1,_in2
    mov   R2,_in1
; __asm statement expansion
    MOV A, R2
    ADD A, R1
    MOV R1, A
; Code generated by C compiler
    mov   _out, R1
_main:
    mov   _in1,#3
    mov   _in2,#4
```

Example 7: 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 char foo( char par1, char par2, char * par3 )
{
  int retvalue;

  __asm(
    "dec     %1\n\t"
    "mov     A,%2\n\t"
    "add     A,%1\n\t"
    "mov     A%0,%5\n\t"
    "mov     @%0,A"
    : "=&s" (retvalue), "=r" (par1), "=r" (par2)
    : "1" (par1), "2" (par2), "r" (par3)
  );
  return retvalue;
}

char result,parm;

void func(void)
{
  result = foo( 100, 100, &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     R0,#100
  mov     AR2,R0
  lea     R3,#_parm

  dec     R0              ; R0 contains 99
  mov     A,R2            ; A  contains 100
  add     A,R0            ; A  contains 199
  mov     AR1,R3
  mov     @R1,A

  mov     _result,R1
```

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 100. Therefore the compiler can optimize and assign the same register to %1 and %2. This would have given an unexpected result.

```
__asm(
    "dec     %1\n\t"
    "mov     A,%2\n\t"
    "add     A,%1\n\t"
    "mov     A%0,%3\n\t"
```

```
    "mov    @%0,A"
    : "=&s" (retvalue)
    : "r" (par1), "r" (par2), "r" (par3)
);
```

Generated assembly code:

```
  mov    R0,#_parm
  mov    R2,#100

  dec    R2            ; R2 contains 99
  mov    A,R2          ; A  contains 99
  add    A,R2          ; same register R2, but is expected read-only
  mov    AR1,R0
  mov    @R1,A

  mov    _result,R1  ; contains unexpected result
```

1.5. 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. 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'.

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. 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.

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).

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`.

See also `#pragma section`.

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 2016 TASKING 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`.

1.6. 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 **c51 --help=pragmas** to get a list of all supported pragmas. Pragmas marked with (*) support a label prefix.

alias *symbol=defined_symbol*

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

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

This pragma is used to mark the macros "false" and "true" from the library header file `stdbool.h` as "essentially BOOLEAN", which is a concept from the MISRA C:2012 standard.

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. Note however that constant data in `__rom` space is never cleared. So,

```
__rom int i;          /* always uninitialized */
__rom const int j;  /* always uninitialized */
```

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

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

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

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

extend {*size* | default | restore} (*)

Specify the maximum amount of internal RAM to be used for pseudo registers.

See C compiler option **--extend** and Section 1.8.1, *Automatic Variables*.

extern *symbol*

Normally, when you use the C keyword extern, the compiler generates an .EXTRN 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 .EXTRN directive out.

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

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**.

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

See Section 1.11.4, *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 enabled the inlining optimization (C compiler option **--optimize=+inline**).

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

linear_switch / jump_switch / binary_switch / smart_switch [default | restore] (*)

With these pragmas you can overrule the compiler chosen switch method:

linear_switch   force jump chain code. A jump chain is comparable with an if/else-if/else-if/else construction.

jump_switch   force jump table code. A jump table is a table filled with jump instructions for each possible switch value. The switch argument is used as an index to jump within this table.

binary_switch   force binary lookup table code. A binary search table is a table filled with a value to compare the switch argument with and a target address to jump to.

smart_switch   let the compiler decide the switch method used

See also Section 1.10, *Switch Statement*.

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 3.7.2, *C Code Checking: MISRA C*.

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

Do not generate interrupt vectors and reference to interrupt handler in run-time library. Same as C compiler option **--no-vector**.

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 3.6, *Compiler Optimizations*.

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

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`.

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

Allocate strings in ROM and RAM. The strings are copied to RAM at startup.

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

Allocate strings in ROM only. Same as C compiler option **--romstrings (-S)**.

section [*type*=*name*] / endsection [default | restore] (*)

Rename sections of the specified *type* or restore default section naming. See Section 1.12, *Section Naming* for more information.

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.

tradeoff {*level* | default | restore} (*)

Specify tradeoff between speed (0) and size (4).

vector_offset {*offset* | default | restore} (*)

Specify base address for interrupt vectors.

See C compiler option **--vector-offset**.

warning [*number*[-*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.

1.7. 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
__BIG_ENDIAN__	Expands to 1. The processor accesses data in big-endian, except for the __sfr space which is little-endian.
__BUILD__	Identifies the build number of the compiler in the format yymmddqq (year, month, day and quarter in UTC).
__C51__	Identifies the compiler. You can use this symbol to flag parts of the source which must be recognized by the **c51** compiler only. It expands to 1.
__CORE_*core*__	A symbol is defined depending on the option **--core=**_core_. The _core_ is converted to uppercase. For example, if **--core=scr2g** is specified, the symbol __CORE_SCR2G__ is defined. When no **--core** is supplied, the compiler defines __CORE_C51__.
__DATE__	Expands to the compilation date: "mmm dd yyyy".
__FILE__	Expands to the current source file name.
__LINE__	Expands to the line number of the line where this macro is called.
__LITTLE_ENDIAN__	Expands to 0. The processor accesses data in big-endian, except for the __sfr space which is little-endian.
__MISRAC_VERSION__	Expands to the MISRA C version used 1998, 2004 or 2012 (option **--misrac-version**). The default is 2004.
__MODEL__	Identifies the memory model for which the current module is compiled. It expands to a single character constant: 's' (small), 'a' (auxiliary page), or 'l' (large).
__PROF_ENABLE__	Always expands to 0 (dynamic profiling is disabled).
__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
__SINGLE_FP__	Always expands to 1 (8051 only has single precision floating-point).
__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.

Macro	Description
__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).
__VX__	Identifies the VX-toolset C compiler. Expands to 1.

Example

```
#ifdef __C51__
/* this part is only compiled for the 8051 VX-toolset compiler */
...
#endif
```

1.8. Variables

1.8.1. Automatic Variables

In non-reentrant functions recursion is not possible, because automatic variables are not allocated on a stack, but in a static area. The static area of a function can be overlaid with that of another function. This saves memory. Depending on the selected memory model the static area for automatics will be allocated in the data, pdata or xdata memory space for the memory models small, auxiliary or large respectively.

In a reentrant function automatic variables are treated the conventional way: dynamically allocated on a stack. As is the case for the static area the place of the stack depends upon the selected memory model, it can be allocated in the data, pdata or xdata memory space.

Although automatic variables are allocated in a static area for non-reentrant functions, they are not the same as local variables (within a function) which are declared to be static by means of the keyword static. When the keyword static is used, a variable will keep its value when a function returns and is called again. This is not the case for automatic variables allocated in the static area, because the area may be overlaid with the static area of another function.

To generate code which is as fast and compact as possible, the compiler tries to place some automatic variables into registers and in the internal RAM (extended virtual registers, also known as pseudo registers). By default, the compiler uses four bytes per function for pseudo registers. You can change this amount by means of the C compiler option **--extend=***size* or `#pragma extend` *size*.

For non-reentrant functions the static area for the pseudo registers will be overlaid, like the static area for automatic variables.

For reentrant functions the area for the pseudo registers is as large as required for the function that uses the most pseudo registers. Reentrant functions save/restore the pseudo registers on the stack, like they are real registers.

The C library is built in such a way that no pseudo registers are used. I.e: it is built with the option **--extend=0**.

1.8.2. Initialized Variables

Non automatic initialized variables use the same amount of space in both ROM and RAM (for all possible RAM memory spaces). This is because the initializers are stored in ROM and copied to RAM at start-up. This is completely transparent to the user. The only exception is an initialized variable residing in ROM, by means of the `__rom` memory type qualifier.

The following examples are for the 8051 VX-toolset compiler in the large memory model.

```
int         i = 100;        /* 2 bytes in ROM, 2 bytes in XDATA */
__rom int   j = 3;          /* 2 bytes in ROM */
char        *p = "TEXT";     /* 7 bytes in ROM, 7 bytes in XDATA:
                               2 bytes for p, 5 bytes for "TEXT" */
__rom  char h[] = "HELP";    /* 5 bytes in ROM */
__data char c = 'a';         /* 1 byte in ROM, 1 byte in DATA */
```

1.8.3. Non-Initialized Variables

In some cases clearing or initialization of global variables at startup is unwanted. For example when memory contents are preserved after power is turned off (see for an example Section 6.7.8, *The Section Layout Definition: Locating Sections*). This can be the case when some RAM is implemented in EEPROM or in a battery powered memory device. To prevent a global variable from being initialized is easy: just do not initialize it. To avoid clearing of non-initialized variables one of the following should be done:

• Define (allocate) these variables in a special C module and compile this module with option **--no-clear**. From Eclipse: From the **Project** menu, select **Properties for**, expand **C/C++ Build**, select **Settings** and open the **Tool Settings** tab, select **C Compiler » Allocation** and disable the option **Clear uninitialized global and static variables**.

• Define (allocate) these variables between `#pragma noclear` and `#pragma clear`.

• Make a separate assembly module, containing the allocation of these variables in a special data section.

1.9. Strings

In this context the word 'strings' means the separate occurrence of a string in a C program. So, array variables initialized with strings are just initialized character arrays, which can be allocated in any memory type, and are not considered as 'strings'.

The 8051 compiler places strings in both ROM and RAM. Where strings in RAM are placed depends on the specified memory model. If you use the 8051 compiler option **--romstrings** or `#pragma romstring`, the compiler places all strings in ROM only. This is useful for single chip applications.

Example without **--romstrings** option:

```
__rom char hello[] = "Hello\n"; /* initialized array in ROM only */
char *world = "world\n";         /* initialized pointer
                                    to string in XDATA */
```

Example with **--romstrings** option:

```
__rom char hello[] = "Hello\n"; /* initialized array in ROM only */
__rom char *world = "world\n";  /* initialized pointer
                                    to string in ROM */
```

Example with `#pragma romstring`:

```
#pragma romstring
__rom char hello[] = "Hello\n"; /* initialized array in ROM only */
__rom char *world = "world\n";  /* initialized pointer
                                    to string in ROM */
#pragma ramstring
```

Strings in library routines

Library routines containing pointer arguments always expect the target memory of these pointers to be the default RAM of the memory model used to make this library. For example:

```
int printf( const char *format, ... );
```

In the large memory model, this means `printf()` expects the address of the format string (the first argument) to have memory type `__xdata`. Therefore, the C startup code of the large memory model copies all strings from ROM to XDATA. So, the statement:

```
printf( "Hello world\n" );
```

is executed correctly, because the 8051 compiler passes the address of the allocated XDATA area (filled at C startup time) to `printf()`.

With the **--romstrings** option specified, the string is put in ROM only and the standard printf/scanf like library routines will fail. You will need to create your own `__rom` qualified versions. All library sources are delivered. For example, you can use the source of `printf()` (in module `printf.c`) to create a `__rom_printf()`. The prototype could be:

```
int __rom_printf( __rom char *format, ... );
```

Modifying string literals

The 8051 accepts that string literals are modifiable when strings are in both ROM and RAM. You can do this with pointers, or even with a construct like:

```
"st ing"[2] = 'r';
```

Of course, when you use the **--romstrings** option this statement is not allowed.

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 binary search table.

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

`#pragma smart_switch` is the default of the compiler. The compiler tries to use the switch method which uses the least space in ROM (table size in ROMDATA plus code to do the indexing). With the C compiler option **--tradeoff** you can tell the compiler to emphasis more on speed than on ROM size.

For a switch with a long type argument, only binary search table code is used.

For an int type argument, a jump table switch is only possible when all case values are in the same 256 value range (the high byte value of all programmed cases are the same).

Especially for large switch statements, the jump table approach executes faster than the binary search 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.

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

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

`#pragma linear_switch`	force jump chain code
`#pragma jump_switch`	force jump table code
`#pragma binary_switch`	force binary search table code
`#pragma smart_switch`	let the compiler decide the switch method used

Using a pragma cannot overrule the restrictions as described earlier.

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

Example:

```
void test(unsigned char val)
{
/* place pragma before the switch statement */

#pragma jump_switch
    switch (val)
    {
        /* use jump table */
    }
}
```

1.11. Functions

Static and Reentrant Functions

For the 8051 VX-toolset functions in C can either be *static* or *reentrant*. In static functions parameters and automatic variables are not allocated on a stack, but in a static area. Reentrant functions use a less efficient virtual dynamic stack which allows you to call functions recursively. With reentrancy, you can call functions at any time, even from interrupt functions. The compiler can overlay parameters and automatics of static functions, but not of reentrant functions.

See also Section 1.8.1, *Automatic Variables*.

You can use the function qualifiers `__static` or `__reentrant` to specify a function as static or reentrant, respectively. If you do not specify a function qualifier, the compiler assumes that those functions are static. If you specify the compiler option **--reentrant** the default for functions without a function qualifier is reentrant.

Example:

```
void f_static( void )
{
    /* this function is by default __static */
}

__reentrant int f_reentrant ( void )
{
    int i;
    /* variable i is placed on a virtual stack */
}
```

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 (a static or reentrant stack depending on the `__reentrant` function qualifier).

The following conventions are used when passing parameters to functions.

Registers available for parameter passing are B.0 .. B.7, R2, R3, R4, R5, R6, and R7. Parameters <= 32 bit are passed in registers:

Parameter Type	Registers used for parameters
1 bit	B.0, B.1, B.2, B.3, B.4, B.5, B.6, B.7
8 bit	R7, R5, R3, R6, R4, R2
16 bit	R67, R45, R23

Parameter Type	Registers used for parameters
32 bit	R4567

The parameters are processed from left to right. The first not used and fitting register is used. Registers are searched for in the order listed above. When a parameter is > 32 bit, or all registers are used, parameter passing continues on the stack. Data on the stack is always byte aligned.

Example with three arguments:

```
func1( int a, long b, int *c )
```

a (first parameter) is passed in registers R67.

b (second parameter) is passed on the stack. (R67 from R4567 is already used)

c (third parameter) is passed in registers R45.

Variable Argument Lists

Functions with a variable argument list must push all parameters after the last fixed parameter on the stack. The normal parameter passing rules apply for all fixed parameters.

Function Return Values

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

C, A, R4, R5, R6 and R7 are used for return values <=32 bit:

Return Type	Register	Description
1 bit	C	carry
8 bit	A	accumulator
16 bit	R67	R6 high byte, R7 low byte
32 bit	R4567	R45 high word, R67 low word

The return registers have an overlap with the parameter registers, which yields more efficient code when passing arguments to child functions.

1.11.2. Stack Usage

The stack consists of a system stack in `__idata`, two virtual dynamic stacks (in `__xdata` and `__pdata`) and three static stacks (static areas in `__data`, `__xdata` and `__pdata`). The system stack and all static stacks grow up, the virtual stacks grow down. The system stack pointer and virtual stack pointers always point to the last valid byte.

The following figures show the layout of the system stack and the virtual stack.
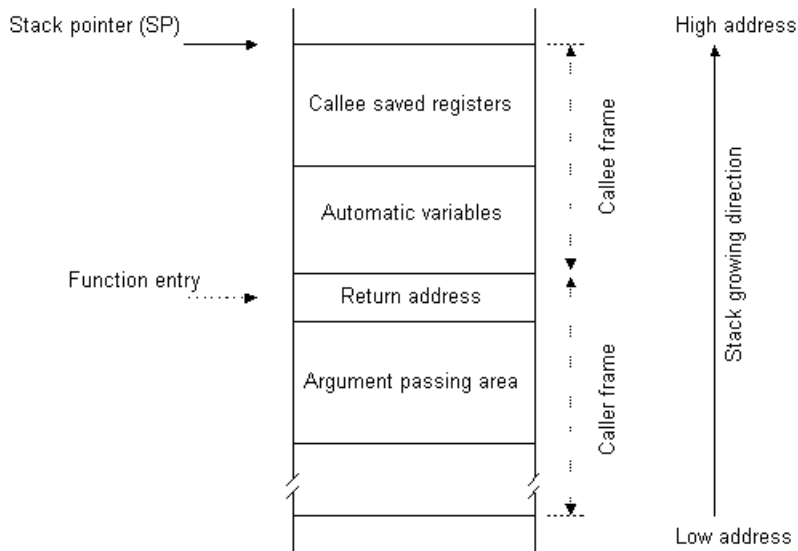
Figure 1.1. System stack layout
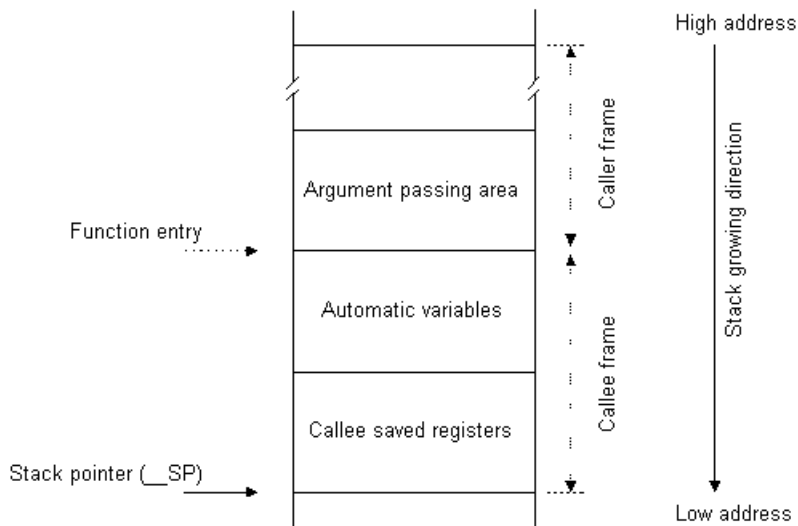
Figure 1.2. Virtual stack layout

The following shows the stack usage per memory model.

Stack usage in small memory model

The following applies to functions implicitly or explicitly qualified `__reentrant`:

Saved value	Stack	Offset (from function entry)	Stack pointer
Parameters	system	- (size of parameters + 1)	SP
Return address	system	-1	SP
Automatic variables	system	+1	SP
Saved registers	system	size of automatic variables + 1	SP

The following applies to functions implicitly or explicitly qualified `__static`:

Saved value	Stack	Offset	Stack pointer
Return address	system	-1	SP
Registers	static stack in `__data`	0	label
Automatic variables	static stack in `__data`	size of registers	label
Parameters	static stack in `__data`	size of registers + size of automatic variables	label

Stack usage in large memory model

The following applies to functions implicitly or explicitly qualified `__reentrant`:

Saved value	Stack	Offset	Stack pointer
Return address	system	-1	SP
Saved registers	virtual stack in `__xdata`	- size of automatic variables and saved registers	__SP
Automatic variables	virtual stack in `__xdata`	- size of automatic variables	__SP
Parameters	virtual stack in `__xdata`	0	__SP

The following applies to functions implicitly or explicitly qualified `__static`:

Saved value	Stack	Offset	Stack pointer
Return address	system	-1	SP
Automatic variables	static stack in `__xdata`	0	label
Parameters	static stack in `__xdata`	size of automatic variables	label
Registers	static stack in `__data`	0	label

Stack usage in auxiliary page memory model

The following applies to functions implicitly or explicitly qualified `__reentrant`:

Saved value	Stack	Offset	Stack pointer
Return address	system	-1	SP
Saved registers	virtual stack in `__pdata`	- size of automatic variables and saved registers	__SP

Saved value	Stack	Offset	Stack pointer
Automatic variables	virtual stack in __pdata	- size of automatic variables	__SP
Parameters	virtual stack in __pdata	0	__SP

Same virtual stack picture as with the large memory model, but then in __pdata instead of in __xdata.

The following applies to functions implicitly or explicitly qualified __static:

Saved value	Stack	Offset	Stack pointer
Return address	system	-1	SP
Automatic variables	static stack in __pdata	0	label
Parameters	static stack in __pdata	size of automatic variables	label
Registers	static stack in __data	0	label

1.11.3. Register Usage

The C compiler uses the general purpose registers and pseudo registers according to the convention given in the following table (for normal functions).

Register	Class	Purpose
A	caller saves	Return value
B	caller saves	Parameter passing
DPL	caller saves	
DPH	caller saves	
R0	caller saves	
R1	caller saves	
R2	caller saves	Parameter passing
R3	caller saves	Parameter passing
R4	caller saves	Parameter passing and return values
R5	caller saves	Parameter passing and return values
R6	caller saves	Parameter passing and return values
R7	caller saves	Parameter passing and return values
PSW	caller saves	Program Status Word register
pseudo registers %__REG+*reg*	callee saves	Automatic variables
SP	dedicated	Stack pointer

The registers are classified: caller saves, callee saves and dedicated.

caller saves   These registers are allowed to be changed by a function without saving the contents. Therefore, the calling function must save these registers when necessary prior to a function call.

callee saves   These registers must be saved by the called function, i.e. the caller expects them not to be changed after the function call.

dedicated   The stack pointer register SP is dedicated.

For interrupt functions (see Section 1.11.5, *Interrupt Functions*), except for the reset vector, the following registers are used for callee saves:

A, B, DPH, DPL, R0 .. R7, PSW, pseudo registers %__REG+*reg*

There are no caller saves registers for interrupt functions.

1.11.4. 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**.

1.11.5. Interrupt Functions

The TASKING C compiler supports a number of function qualifiers and keywords to program interrupt service routines (ISR). An *interrupt service routine* (or: interrupt function, interrupt handler, exception handler) is called when an interrupt event (or: *service request*) occurs.

The difference between a normal function and an interrupt function is that an interrupt function ends with a RETI instruction instead of a RET instruction, and that all registers that might possibly be corrupted during the execution of the interrupt function are saved on function entry (this is called the *interrupt frame*) and restored on function exit.

1.11.5.1. Defining an Interrupt Service Routine: __isr, __interrupt()

You can use the type qualifier `__isr` to declare a function as an interrupt service routine, but this does not bind the function to an interrupt vector. With the function qualifier `__interrupt()` you can bind the function to a specific vector. The function qualifier `__interrupt()` takes one or more vector addresses as argument(s). All supplied vector addresses will be initialized to point to the interrupt function.

The `__interrupt()` function qualifier implies the `__isr` type qualifier.

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

```
void __interrupt(vector_address[, vector_address]...)
isr( void )
{
```

```
...
}
```

The __isr type qualifier must also be used when a pointer to an interrupt function is declared.

For example:

```
void __interrupt( 0x23 ) serial_receive( void )
{   /* __isr is added automatically by __interrupt() */
  ...
}


extern void __isr  external_isr( void );   /* reference to external */
                   /* interrupt function, vector address irrelevant */
void __isr (*pisr)( void ) = external_isr;
                   /* declare pointer to interrupt function */
```

Suppress generation of interrupt vectors

When you define an interrupt service routine, the compiler generates the appropriate interrupt vector, consisting of an instruction jumping to the interrupt function. You can suppress this with the C compiler option **--no-vector** or the #pragma novector.

Specify another vector offset

For certain ROM monitors it is necessary to specify an offset for all interrupt vectors. For this you can use the C compiler option **--vector-offset=***value*. Suppose a ROM monitor has the interrupt table at offset 0x4000. When you compile with --vector-offset=0x4000 interrupt vector 1 (vector address 11) is being located at address 0x400B instead of 0xB.

1.11.5.2. Register Bank Switching: __bank*x* / __nobank

It is possible to assign a new register bank to an interrupt function, which can be used on the processor to minimize the interrupt latency because registers do not need to be pushed on stack. You can switch register banks with the __bank0, __bank1, __bank2 or __bank3 function qualifier. The syntax is:

```
void __interrupt(vector_address[, vector_address]...)
  __bankbanknr
isr( void )
{
...
}
```

When you do not specify a __bank*x* qualifier for an interrupt function, the compiler assumes the default register bank, as set by the compiler option **--registerbank**. In this case the compiler saves the GPRs by using push/pop instructions and generates code to switch to the selected register bank.

With an explicit __bank*x* qualifier, the compiler will only generate code to switch to the selected register bank. The registers R0 - R7 are implicitly saved when the register bank is being switched. When the __bank*x* qualifier is the same as the default, the compiler generates a warning.

Example:

```
#define __INTNO(nr) ((8*nr)+3)  /* use number instead of vector address */

__interrupt(__INTNO(1)) __bank2 void timer(void);
```

The compiler places a long-jump instruction on the vector address 11 of interrupt number 1, to the `timer()` routine, which switches the register bank to bank 2 and saves some more registers. When `timer()` is completed, the extra registers are popped, the bank is switched back to the original value and a RETI instruction is executed.

You can call another C function from the interrupt C function. However, this function must be compiled with the same `__bank`*x* qualifier, because the compiler generates code which uses the addresses of the registers R0-R7. Therefore, the `__bank`*x* qualifier is also possible with normal C functions (and their prototype declarations).

Example:

Suppose `timer()`, from the previous example, is calling `get_number()`. The function prototype (and definition) of `get_number()` should contain the correct`__bank`*x* qualifier.

```
__bank2 int get_number( void );
```

Register bank independent code generation

In order to generate efficient code the compiler uses absolute register addresses in its code generation. For example, since there is no instruction to move a register to a register, the compiler will use a direct addressing mode: "MOV Rn,direct". In the second operand the absolute address of a register will be used.

The absolute address of a register depends upon the selected register bank. Sometimes this dependency is unwanted, for example when a function is called from both the main thread and an interrupt thread. If both threads use different register banks, they cannot call a function that uses absolute register addresses. To overcome this, you can instruct the compiler to generate the code for a function in a register bank independent way. To do this, you can use the `__nobank` qualifier (or use compiler option **--registerbank=none**).

When the code in an interrupt function needs to be generated in a register bank independent way, the compiler will always push/pop the used GPRs. The used register bank will not be switched in this case.

Example:

```
__nobank int func( int x )
{
    /* this function can be called from any function
       independent of its register bank */
    return x+1;
}
__bank1 void f1( void )
{
    func( 1 );
}
```

```
__bank0 void main( void )
{
    func( 0 );
}
```

1.11.5.3. Reset Vector

The compiler treats the reset vector (`__interrupt( 0 )`) as a special case. For this vector the compiler will never push/pop any registers. Also the PSW register will not be saved/restored before initializing it with the selected register bank. Furthermore, the compiler will not warn when an explicit __bank*x* qualifier is the same as the default.

Because of the special treatment, the reset vector cannot be combined with other interrupt vectors. E.g: `__interrupt( 11, 0, 19 )` is not allowed.

1.11.5.4. Interrupt Frame: __frame()

With the function qualifier `__frame()` you can specify which registers and SFRs must be saved for a particular interrupt function. Only the specified registers will be pushed and popped from the stack. If you do not specify the function qualifier `__frame()`, the C compiler determines which registers must be pushed and popped. The syntax is:

```
void __interrupt(vector_address[, vector_address]...)
      __frame(reg[, reg]...) isr( void )
{
...
}
```

The *reg* can be any register defined as an SFR. The compiler generates a warning about registers that are not listed in `__frame()` but are used in the interrupt function. When the compiler would save GPRs using push/pop instructions it will warn about missing GPRs in `__frame()` also. The compiler does not generate a warning when an explicit __bank*x* qualifier is used because the compiler would not push/pop GPRs itself.

Example:

```
void __interrupt(0x10) __frame(A,R0,R1) foo (void)
{
 ...
}
```

Normally when an interrupt function is called, all registers in the default register bank that are (or could be) used in the interrupt function are saved on the stack so the registers are available for the interrupt routine. After returning from the interrupt routine, the original values are restored from the stack again.

1.11.6. 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 TASKING VX-toolset for 8051 C compiler recognizes the following intrinsic functions:

__alloc

```
void * volatile __alloc( __size_t size );
```

Allocate memory. Returns a pointer to space in external memory of *size* bytes length. 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.

__dotdotdot__

```
char * volatile __dotdotdot__( void );
```

Variable argument '...' operator. Used in library function `va_start()`. Returns the stack offset to the variable argument list.

__free

```
void volatile __free( void *p );
```

Deallocates the memory pointed to by `p`. `p` must point to memory earlier allocated by a call to `__alloc()`.

__getbit

```
__bit __getbit( operand, bitoffset );
```

Returns the bit at `bitoffset` (range 0-7 for a char, 0-15 for an int or 0-31 for a long) of the bit-addressable `operand` for usage in bit expressions. `bitoffset` must be an integral constant expression.

Example:

```
__bdata unsigned char byte;
int i;

if ( __getbit( byte, 3 ) )
    i = 1;
```

__putbit

```
void __putbit( __bit value, operand, bitoffset );
```

Assign `value` to the bit at `bitoffset` (range 0-7 for a char, 0-15 for an int or 0-31 for a long) of the bit-addressable `operand`. `bitoffset` must be an integral constant expression.

Example:

```
__bdata unsigned int word;

__putbit( 0, word, 10 );
```

__nop

```
void __nop( void );
```

Generates a NOP instruction.

__rol

```
unsigned char __rol( unsigned char operand,
                     unsigned char count );
```

Use the RL instruction to rotate `operand` left `count` times. Returns the rotated value.

__ror

```
unsigned char __ror( unsigned char operand,
                     unsigned char count );
```

Use the RR instruction to rotate `operand` right `count` times. Returns the rotated value.

__testclear

```
__bit __testclear( __bit *semaphore );
```

Read and clear `semaphore` using the JBC instruction. Returns 0 if `semaphore` was not cleared by the JBC instruction, 1 otherwise.

Example:

```
__bit b;
unsigned char c;

if ( __testclear( &b ) ) /* JBC instruction */
    c=1;
```

__vsp__

```
__bit __vsp__( void );
```

Virtual stack pointer used. Used in library function `va_arg()`. Returns 1 if the virtual stack pointer is used, 0 otherwise.

1.12. Section Naming

The C compiler generates sections and uses a combination of the memory type and the object name as section names. The memory types are: code, rom, bit, bdata, data, idata, pdata and xdata. See also Section 1.2.1, *Memory Type Qualifiers*. The section names are independent of the section attributes such as clear, init, and romdata.

Section names are case sensitive. By default, the sections are not concatenated by the linker. This means that multiple sections with the same name may exist. At link time sections with different attributes can be selected on their attributes. The linker may remove unreferenced sections from the application.

You can rename sections with a pragma or with a command line option. The syntax is the same:

--rename-sections=[*type***=**]*format_string*[**,**[*type***=**]*format_string*]...

```
#pragma section [type=]format_string[,[type=]format_string]...
```

With the memory *type* you select which sections are renamed. The matching sections will get the specified format string for the section name. The format string can contain characters and may contain the following format specifiers:

{attrib}	section attributes, separated by underscores
{module}	module name
{name}	object name, name of variable or function
{type}	section type

The default compiler generated section names are {type}_{name}.

It is not possible to change the name of overlay sections, max sections and interrupt vector table sections.

Some examples (file test.c):

```
#pragma section data={module}_{type}_{attrib}
__data int x;
/* Section name: test_data_data_clear */

#pragma section data=_8051_{module}_{name}
__data int status;
/* Section name: _8051_test_status */

#pragma section pdata=RENAMED_{name}
__pdata int barcode;
/* Section name: RENAMED_barcode */
```

With #pragma endsection the naming convention of the previous level is restored, while with #pragma section default the default section naming convention is restored. Nesting of pragma section/endsection pairs will save the status of the previous level.

Examples (file example.c)

```
__data char a;          // allocated in 'data_a'
#pragma section data=MyData1
__data char b;          // allocated in 'MyData1'
#pragma section data=MyData2
__data char c;          // allocated in 'MyData2'
#pragma endsection
__data char d;          // allocated in 'MyData1'
#pragma endsection
__data char e;          // allocated in 'data_e'
```

Chapter 2. Assembly Language

This chapter describes the most important aspects of the TASKING assembly language. For a complete overview of the architecture you are using, refer to the target's Core Reference Manual.

2.1. 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, dollar ($) and underscore characters (_). The first character cannot be a digit or a $. 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: jmp 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 2.3, *Operands of an Assembly Instruction*. The instructions are described in the target's Core Reference Manual.
	The instruction can also be a so-called 'generic instruction'. Generic instructions are pseudo instructions (no instructions from the instruction set). Depending on the situation in which a generic instruction is used, the assembler replaces the generic instruction with appropriate real assembly instruction(s). For a complete list, see Section 2.10, *Generic Instructions*.
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 2.9, *Assembler Directives and Controls*.
macro_call	A call to a previously defined macro. It must not start in the first column. See Section 2.8, *Macro Preprocessing*.
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 2.9, *Assembler Directives and Controls*.

2.2. 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 2.7.3, *Expression Operators*. Other special assembler characters are:

Character	Description
;	Start of a comment
;;	Unreported comment delimiter
\	Line continuation character
%	Start of a built-in assembly function, or a macro call
*	Literal character, used in `%*DEFINE`
”	String constants delimiter
'	String constants delimiter

Character	Description
$	Location counter substitution
#	Immediate addressing

Note that macro operators have a higher precedence than expression operators.

2.3. 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 2.4, *Symbol Names*. Symbols can also occur in expressions.
register	Any valid register as listed in Section 2.5, *Registers*.
expression	Any valid expression as described in Section 2.7, *Assembly Expressions*.
address	A combination of *expression*, *register* and *symbol*.

2.4. 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 2.4.1, *Predefined Preprocessor Symbols*.

Labels

Symbols used for memory locations are referred to as labels.

Reserved symbols

Symbol names and other identifiers beginning with a period (.) are reserved for the system (for example for directives or section names). Instructions 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
.DEFINE     ; reserved directive name
```

2.4.1. 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
__BUILD__	Identifies the build number of the assembler in the format yymmddqq (year, month, day and quarter in UTC).
__C51__	Identifies the assembler. You can use this symbol to flag parts of the source which must be recognized by the **as51** assembler only. It expands to 1.
__CORE_	Expands to a string with the core name depending on the option **--core=**_core_. For example, if **--core=scr2g** is specified, the symbol __CORE__ expands to scr2g.
__CORE_*core*__	A symbol is defined depending on the option option **--core=**_core_. The _core_ is converted to uppercase. For example, if **--core=scr2g** is specified, the symbol __CORE_SCR2G__ is defined as 1, otherwise the symbol is defined as 0. When no **--core** is supplied, the assembler defines __CORE_C51__ as 1.
__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.
__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 @defined('__C51__')
  ; this part is only for the 8051 assembler
...
.endif
```

2.5. Registers

The following register names, either uppercase or lowercase, should not be used for user-defined symbol names in an assembly language source file:

```
A    C    DPTR
R0   R1   R2   R3   R4   R5   R6   R7
AR0  AR1  AR2  AR3  AR4  AR5  AR6  AR7
```

The following special function registers should also not be used as symbol names in an assembly language source file. However it is allowed to redefine them.

```
ACC  B    DPH  DPL  PSW  SP
AC   CY   F0   F1   P    OV   RS0  RS1
```

2.6. 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 file (*.sfr) as symbol names for use with the compiler and assembler. The assembler reads the SFR file with the command line option **--sfr-file** . If you use Eclipse or the control program you can specify that the SFR file should be included based on the selected processor automatically (**--asm-sfr-file**). The format of the SFR file is exactly the same as the include file for the C compiler. For more details on the SFR files see Section 1.2.5, *Accessing Hardware from C: __sfr, __bsfr*. Because the SFR file format uses C syntax and the assembler has a limited C parser, it is important that you only use the described constructs.

Example use in assembly (with option **--sfr-file=regtc26x.sfr**):

```
  mov   SCON,#0x88      ; use of SFR name
  setb  SCR_P00_IN_3    ; use of bit name
  gjnb  SCR_P00_IN_4,_2
  clr   SCR_P00_IN_3
_2:
  setb  TCON_IE1        ; use of bit name
```

Without an SFR file the assembler only knows the registers and SFRs as specified in Section 2.5, *Registers*.

Built into the assembler are a number of symbol definitions for various 8051 addresses in bit and data memory space. These symbols are treated by the assembler as if they were defined with the .BIT or .DATA directives.

Bit addresses

Symbol	Address	Symbol	Address
P	0xD0	RS1	0xD4
F1	0xD1	F0	0xD5
OV	0xD2	AC	0xD6
RS0	0xD3	CY	0xD7

Data addresses AURIX family/generic 8051

Symbol	Address	Symbol	Address
SP	0x81	PSW	0xD0
DPL	0x82	ACC	0xE0
DPH	0x83	B	0xF0

Data addresses AURIX 2G family SCR

Symbol	Address	Symbol	Address
SP	0xD4	PSW	0xD0
DPL	0xD5	ACC	0xE0
DPH	0xD6	B	0xDA

Note that the B register for the AURIX 2G family SCR is not bit addressable, whereas the B register for the AURIX family/generic 8051 is bit addressable.

2.7. 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 values), and any combination of integers 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.

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.

2.7.1. 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 and suffixes can be used in either lowercase or uppercase.

Base	Description	Example
Binary	A **0b** prefix followed by binary digits (0,1). Or use a **b** suffix	`0b1101` `11001010b`
Octal	Octal digits (0-7) followed by a **o** or **q** suffix	`777o`

Hexadecimal	A **0x** prefix followed by a hexadecimal digits (0-9, A-F, a-f). Or use a **h** suffix	`0x12FF`
		`0x45`
		`0fa10h`

Decimal	Decimal digits (0-9), optionally followed by a **d**	`12` `1245d`

2.7.2. Strings

ASCII characters, enclosed in single (') or double (") quotes constitute an ASCII string. 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 `.DB`, `.DW` or `.DL` assembler directive; in that case all characters result in a constant value of the specified size. Null strings have a value of 0.

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
```

2.7.3. 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.

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.
	! NOT	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.
	HIGH	high byte	Returns the high byte of the operand ((operand >> 8)&0xFF).
	LOW	low byte	Returns the low byte of the operand (operand & 0xFF).
	type	type cast	Any of the valid assembler symbol types can be used as a type cast operator.
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.
	% MOD	modulo	Used with integers, this operator yields the remainder from the division of the first operand by the second. Used with floating-point operands, this operator applies the following rules: Y % Z = Y if Z = 0 Y % Z = X if Z <> 0, where X has the same sign as Y, is less than Z, and satisfies the relationship: Y = integer * Z + X
	+	addition	Yields the sum of its operands.
	-	subtraction	Yields the difference of its operands.
Shift	<< SHL	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.

Type	Operator	Name	Description
	>> SHR	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	< LT	less than	Returns an integer 1 if the indicated condition is TRUE or an integer 0 if the indicated condition is FALSE. In either case, the memory space attribute of the result is N 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.
	<= LE	less than or equal	
	> GT	greater than	
	>= GE	greater than or equal	
	== EQ	equal	
	!= NE	not equal	
Bit and Bitwise	.	bit position	Specify bit position (right operand) in a bit addressable byte or word (left operand).
	& AND	AND	Integer only. Yields the bitwise AND function of its operand.
	| OR	OR	Integer only. Yields the bitwise OR function of its operand.
	^ XOR	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 function, but can be used in any expression.

2.7.4. Symbol Types and Expression Types

Symbol Types

The type of a symbol is determined upon its definition by the section in which it is defined. The following table shows the symbol types that are available.

Symbol type	Type of section where symbol is defined
BIT	bit address space
CODE	code address space
DATA	direct addressable data

Symbol type	Type of section where symbol is defined
IDATA	indirect addressable data
PDATA	auxiliary external data space
XDATA	external data space

It is also possible to explicitly define the symbol's type with the `.BIT`, `.CODE`, `.DATA`, `.IDATA` and `.XDATA` directive and with the `.EXTRN` directive. Labels not on the same line as the directive still are assigned the type for that directive if they immediately precede the directive:

```
codesect   .segment code
           .rseg codesect
mylabel:   ; this label gets the CODE type
           .dw  1
```

When you make a symbol global with the `.PUBLIC` directive, the symbol's type will be stored in the object file. The `.EXTRN` directive used for importing the symbol in another module must specify the same type.

Symbols defined with `.EQU` or `.SET` inherit the type of the expression. The result of an expression is determined by the type of symbols used in the expression.

Type Checking

When you use a symbol or expression as an operand for an instruction, the assembler will check if the type of this symbol or expression is valid for the used instruction. If it is not valid, the assembler will issue an error. For generic instructions the assembler uses the symbol type to select the smallest instruction.

Expression Types

When evaluating an expression, the result of the expression is determined by the operands of the expression and the operators. The section type NUMBER is used for expressions representing a typeless number. The section type of an expression involving more than one operand is assigned according to the following rules:

1. The section type of a unary operation (+, -, NOT, LOW, HIGH) will be the same as the section type of the operand.

2. The section type of a binary + or - operation is NUMBER, unless one of the operands has type NUMBER, in which case the section type will be the type of the other operand.

3. The section type of the binary operations except + and - will be NUMBER.

2.8. Macro Preprocessing

The assembler has a built-in macro preprocessor which is compatible with Intel's syntax for the 8051 macro processing language (MPL).

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.

2.8.1. Defining and Calling Macros

The first step in using a macro is to define it. Every macro consists of a *macro_name* with optional *arguments* and a *macro_body*. The *macro_name* defines the name used when the macro is called; the *macro_body* contains the code or instructions to be inserted when the macro is called.

A macro definition takes the following form:

```
%[*]DEFINE(macro_name[(argument[,argument]...)]) [LOCAL local_list]
(
    macro_body
    ...
)
```

The '**%**' character signals a macro call. This character is called the 'metacharacter' and can also be redefined (see macro preprocessing function `%METACHAR()`). The '*' is the optional literal character. When you define a macro using the literal character '*', as shown above, macro calls contained in the body of the macro are not expanded until the macro is called.

The *macro_name* must be an identifier conform to the following rules:

- The macro name starts with an uppercase or lowercase alphabetical character (a-z, A-Z), question mark '? or underscore '_'.

- The remaining part of the name can also contain digits.

- Only the first 31 characters of a macro identifier are recognized as the unique identifier name. Uppercase and lowercase characters are not distinguished in a macro identifier.

The *macro_body* may contain calls to other macros. If so, the return value is actually the fully expanded macro body, including the return values of the call to other macros. The macro call is re-expanded each time it is called.

With the optional *argument* list you can pass information to the macro body. You can see them as variables. Each argument must be a unique macro identifier, but it may be the same as other argument names to other macros since it has no existence outside the macro definition. Argument names may also be the same as the names of other user macros or of macro functions. Note, however that in this case the macro or function cannot be used within the macro body, since its name would be recognized as a parameter instead. To reference an argument within the macro body, use its name preceded by the metacharacter (by default the '**%**' character).

Arguments are separated and surrounded by a delimiter. Typically these are parentheses and commas, but you can use other delimiters as well. See Section 2.8.4, *Macro Delimiters* for more information.

Macros can optionally contain local symbols. For each symbol in the *local_list*, the assembler will replace each symbol by a unique assembly-time symbol each time the macro is called. See Section 2.8.2, *Local Symbols in Macros* for more information.

Once a macro has been created, it may be redefined by a second call to DEFINE. Note, however that a macro should not redefine itself within its body.

Calling a macro

To call a macro, you use the '**%**' character followed by the name of the macro (the literal character '*' is only admissible for defined macros whose call is passed to another macro as an argument; example: `%M1(%*M2)`. The preprocessor removes the call and inserts the return value of the call. If the macro body contains any call to other macros, they are replaced with their return values.

`%macro_name[(argument[,argument]...)] [;comment]`

Example 1: macro definitions without arguments

Macro definition at the top of the program:

```
%*DEFINE (MOVE)
(    MOV  A, @R1
     MOV  @R2, A
)
```

The macro call as it appears in the program:

```
     MOV  R1, #1
  %MOVE   ; <-- macro call preceded by four spaces
```

The program as it appears after the preprocessor of the assembler made the following expansion, where the first expanded line is preceded by the four spaces preceding the call:

```
     MOV  R1, #1
        MOV  A, @R1
     MOV  @R2, A
```

Macro definition at the top of the program:

```
%*DEFINE (ADD5)
(    MOV  R0, #5
     MOV  R5, @R2
     ADD  R5, R0
)
```

The macro call as it appears in the original program body:

```
     MOV  R5, #2
%ADD5
```

The program after the macro expansion:

```
MOV  R5, #2
MOV  R0, #5
MOV  R5, @R2
ADD  R5, R0
```

Macro definition at the top of the program:

```
%*DEFINE (MOVE_AND_ADD) (
     %MOVE
     %ADD5
)
```

The macro call as it appears in the body of the program:

```
     MOV  R1, #1
%MOVE_AND_ADD
```

The program after the macro expansion:

```
     MOV  R1, #1

     MOV  A, @R1
     MOV  @R2, A

     MOV  R0, #5
     MOV  R5, @R2
     ADD  R5, R0
```

Example 2: macro definition with arguments

The example below shows the definition of a macro with three arguments: SOURCE, DEST and COUNT. The macro produces code to copy any number of words from one part of memory to another.

```
%*DEFINE (MOVE_ADD_GEN(SOURCE,DEST,COUNT))
(    MOV  R1, #%SOURCE
     MOV  R0, #%DEST
     MOV  R7, #%COUNT
     MOV  A, @R1
     MOV  @R0, A
     INC  R1
     INC  R0
     DJNZ R7, ($-4)
)
```

A simple call to a macro defined above might be:

```
%MOVE_ADD_GEN( 10, 24, 8 )
```

The above macro call produces the following code:

```
      MOV  R1, #10
      MOV  R0, #24
      MOV  R7, #8
      MOV  A, @R1
      MOV  @R0, A
      INC  R1
      INC  R0
      DJNZ R7, ($-4)
```

2.8.2. Local Symbols in Macros

If we used a fixed label instead of the offset ($-4) in the previous example, the macro using the fixed label can only be called once, since a second call to the macro causes a conflict in the label definitions at assembly time. The label can be made a parameter and a different symbol name can be specified each time the macro is called.

A preferable way to ensure a unique label for each macro call is to put the label in a *local_list*.

The syntax for the LOCAL construct in the DEFINE function is shown below.

```
%[*]DEFINE(macro_name[(argument[,argument]...)]) [LOCAL local_list]
(
    macro_body
    ...
)
```

The *local_list* construct allows you to use macro identifiers to specify assembly-time symbols. Each use of a LOCAL symbol in a macro guarantees that the symbol will be replaced by a unique assembly-time symbol each time the symbol is called.

The macro preprocessor increments a counter once for each symbol used in the list every time your program calls a macro that uses the LOCAL construct. Symbols in the *local_list*, when used in the macro body, receive a two to five digit suffix that is the hexadecimal value of the counter. The first time you call a macro that uses the LOCAL construct, the suffix is '00'.

The *local_list* is a list of valid macro identifiers separated by spaces. Since these macro identifiers are not parameters, the LOCAL construct in a macro has no effect on a macro call.

To reference local symbols in the macro body, they must be preceded by the metacharacter (by default the '**%**' character). The symbol LOCAL is not reserved; a user symbol or macro may have this name.

The next example shows a macro definition that uses a LOCAL list.

Example

```
%*DEFINE (MOVE_ADD_GEN(SOURCE,DEST,COUNT)) LOCAL LAB
(     MOV  R1, #%SOURCE
      MOV  R0, #%DEST
      MOV  R7, #%COUNT
%LAB:
      MOV  A, @R1
```

```
      MOV  @R0, A
      INC  R1
      INC  R0
      DJNZ R7, %LAB
)
```

A simple call to a macro defined above might be:

```
%MOVE_ADD_GEN( 50, 100, 24 )
```

The above macro call might produce the following code (if this is the eleventh call to a macro using a LOCAL list):

```
      MOV  R1, #50
      MOV  R0, #100
      MOV  R7, #24
LAB0A:
      MOV  A, @R1
      MOV  @R0, A
      INC  R1
      INC  R0
      DJNZ R7, LAB0A
```

> Any macro identifier can be used in a *local_list*. However, if long identifier names are used, they should be restricted to 29 characters. Otherwise, the label suffix may cause the identifier to exceed 31 characters and these would be truncated.

2.8.3. Built-in Macro Preprocessing Functions

The macro preprocessor part of the assembler has several built-in or predefined macro functions. These built-in functions perform many useful operations that are difficult or impossible to produce in a user-defined macro.

We have already discussed one of these built-in functions, DEFINE. DEFINE creates user-defined macros. DEFINE does this by adding an entry in the macro preprocessor's tables of macro definitions. Each entry in the tables includes the macro name of the macro, its parameter list, its local list and its macro body. Entries for the built-in functions are present when the macro preprocessor begins operation.

Other built-in functions perform numerical and logical expression evaluation, affect control flow of the macro preprocessor, manipulate character strings, and perform console I/O.

Overview of macro preprocessing functions

Function	Description
%'*text*' %'*text end-of-line*	Comment function
%*n* *text*	Escape function: prevent macro expansion of text of *n* characters long

Function	Description
`%(`*text*`)`	Bracket function: prevent macro expansion
`%{`*text*`}`	Group function: ensure macro expansion
`%EQS()`,`%NES()`,`%LTS()`,`%LES()`, `%GTS()`,`%GES()`	String comparing functions
`%ERROR()`,`%FATAL()`	Generate user error message or fatal error message
`%EXIT`	Terminate expansion of the most recently called user defined macro
`%__FILE__`,`%__LINE__`	File/line info functions
`%IF()`	Conditional control flow
`%IFDEF()`,`%IFNDEF()`	Test if a macro is defined or not
`%IN()`,`%OUT()`	Input/output functions
`%INCLUDE()`	Include a file
`%LEN()`	Return the length of a string
`%MATCH()`	Define macro identifiers
`%METACHAR()`	Redefine the metacharacter '%'
`%OPTION()`	Call a command line option from within the source file
`%SET()`,`%EVAL()`	Calculating functions
`%SUBSTR()`	Return part of a string
`%UNDEF()`	Undefine a previously defined macro or built-in function
`%WHILE()`,`%REPEAT()`	Control looping functions

Comment function: %'

Syntax

```
%'text'
```

or:

```
%'text  end-of-line
```

Description

The macro processing language can be very subtle, and the operation of macros written in a straightforward manner may not be immediately obvious. Therefore, it is often necessary to comment macro definitions.

The comment function always evaluates to the null string. Two terminating characters are recognized: the apostrophe **'** and the end-of-line (line-feed character, ASCII 0AH). The second form of the call allows macro definitions to be spread over several lines, while avoiding any unwanted end-of-lines in the return value. In either form of the comment function, the text or comment is not evaluated for macro calls.

The literal character '*' is not accepted in connection with this function.

Example

```
%*DEFINE (MOVE_ADD_GEN(SOURCE,DEST,COUNT)) LOCAL LAB
(    MOV  R1, #%SOURCE %'This is the source address'
     MOV  R0, #%DEST %'This is the destination'
     MOV  R7, #%COUNT %'%COUNT must be a constant'
%LAB:    %'This is a local label.
%'End of line is inside the comment!
     MOV  A, @R1
     MOV  @R0, A
     INC  R1
     INC  R0
     DJNZ R7, %LAB
)
```

Call the above macro:

```
%MOVE_ADD_GEN( 50, 100, 24 )
```

Return value from above call:

```
     MOV  R1, #50
     MOV  R0, #100
     MOV  R7, #24
LAB0A:
     MOV  A, @R1
     MOV  @R0, A
     INC  R1
     INC  R0
     DJNZ R7, LAB0A
```

Note that the comments that were terminated with the end-of-line removed the end-of-line character along with the rest of the comment.

The metacharacter is not recognized as flagging a call to the macro preprocessor when it appears in the comment function.

Escape function: %n

Syntax

```
%n  text
```

Description

The escape function prevents the macro preprocessor from processing a text string of *n* characters long, where *n* is a decimal digit from 0 to 9. The escape function is useful for inserting a metacharacter as text, adding a comma as part of an argument, or placing a single parenthesis in a character string that requires balanced parentheses.

The literal character '*' is not accepted in connection with this function.

Example

```
Before Macro Expansion      After Macro Expansion

;Average of 20%1%          ->;Average of 20%

%DTCALL(JAN 21%1, 2017,    -> JAN 21, 2017
        AUG 14%1, 2017)    -> AUG 14, 2017

%MYCALL(1%1) Option 1,     -> 1) Option 1
        2%1) Option 2,     -> 2) Option 2
        3%1) Option 2)     -> 3) Option 3
```

The first example adds a literal '%' in the text. The second example keeps the date as one actual parameter adding a literal ','. The third example adds a literal right parenthesis ')' to each parameter.

Related Information

Bracket function %( )

Bracket function: %( )

Syntax

`%(`*text*`)`

Description

The bracket function prevents all macro preprocessor expansion of the *text* contained within the parentheses. However, the escape function, the comment function, and the parameter substitution are still recognized. Since there is no restriction for the length of the text within the bracket function, it is usually easier to use than the escape function.

The literal character '*' is not accepted in connection with this function.

Example

```
%*DEFINE (DEFW(LIST,NAME))
(    %NAME  .DW  %LIST)
```

The macro DEFW expands .DW statements, where the variable `LIST` represents the first parameter and the expression `NAME` represents the second parameter.

The following expansion should be obtained by the call:

```
PHONE .DW 0x198, 0x3D, 0xF0
```

If the call in the following form:

```
%DEFW(0x198, 0x3D, 0xF0, PHONE)
```

occurs, the macro preprocessor would interpret the first argument (0x198) as LIST and everything after the first comma as the second parameter, since the first comma would be interpreted as the delimiter separating the macro parameters.

In order to change this method of interpretation, all tokens that are to be combined for an individual parameter must be identified as a parameter string and set in a bracket function:

```
%DEFW(%(0x198, 0x3D, 0xF0), PHONE)
```

This way the bracket function prevents the string '198H, 3DH, 0F0H' from being evaluated as separate parameters.

Related Information

Escape function %*n*

Group function: %{ }

Syntax

`%{`*text*`}`

Description

The group function does the opposite of the bracket function, it ensures that the text is expanded. The resulting string is then interpreted itself like a macro command. This allows for definition of complex recursive macros. Another useful application of the group function is to separate macro identifiers from surrounding, possibly valid identifier characters.

The literal character '*' is not accepted in connection with this function.

Example

```
%define(TEXTA)(Text A)
%define(TEXTB)(Text B)
%define(TEXTC)(Text C)

%define(SELECT)(B)

%{TEXT%SELECT}
```

The contents of the group function, TEXT%SELECT, expands to TEXTB, which on its turn is expanded as %TEXTB resulting in Text B.

```
%define(op)(add)

%{op}_and_move
```

The group function ensures that the macro op is expanded. Without it, op_and_move would be seen as the macro identifier.

%ERROR, %FATAL

Syntax

%ERROR(*text***)**
%FATAL(*text***)**

Description

With these built-in functions you can generate a user error or fatal error message.

You can use the %ERROR function to trigger a user error 'E 100'. Macro preprocessing will continue after the %ERROR function. The %ERROR function expands to the null string.

You can use the %FATAL function to trigger a user fatal error 'F 101'. Macro preprocessing will stop directly after the %FATAL function, and the program will exit with value 1. The %FATAL function expands to the null string.

Example

```
%IFNDEF(TEMP)
THEN
    (%ERROR(Macro TEMP not defined))
ELSE
    (%FATAL(Macro TEMP is defined))
FI
```

Related Information

$MESSAGE assembler control

%EQS, %NES, %LTS, %LES, %GTS, %GES

Syntax

```
%EQS(arg1,arg2)
%NES(arg1,arg2)
%LTS(arg1,arg2)
%LES(arg1,arg2)
%GTS(arg1,arg2)
%GES(arg1,arg2)
```

Description

These string comparison functions compare two text arguments and return a logical value based on that comparison. If the function evaluates to 'TRUE', then it returns the character string '0ffffH'. If the function evaluates to 'FALSE', then it returns '00H'. Both arguments may contain macro calls.

Function	Description
%EQS	Equal. TRUE if both arguments are identical.
%NES	Not equal. TRUE if arguments are different in any way.
%LTS	Less than. TRUE if first argument has a lower value than second argument.
%LES	Less than or equal. TRUE if first argument has a lower value than second argument or if both arguments are identical.
%GTS	Greater than. TRUE if first argument has a higher value than second argument.
%GES	Greater than or equal. TRUE if first argument has a higher value than second argument, or if both arguments are identical.

Before these functions perform a comparison, both strings are completely expanded. Then the ASCII value of the first character in the first string is compared to the ASCII value of the first character in the second string. If they differ, then the string with the higher ASCII value is to be considered to be greater. If the first characters are the same, the process continues with the second character in each string, and so on. Only two strings of equal length that contain the same characters in the same order are equal.

Example

```
%EQS(ABC,ABC)    ->   0ffffH (TRUE)
```

The character strings are identical.

```
%EQS(ABC, ABC)   ->   00H (FALSE)
```

The space after the comma is part of the second argument

```
%LTS(CBA,cba)    ->   0ffffH (TRUE)
```

The lowercase characters have a higher ASCII value than uppercase.

```
%GES(ABC,ABC)    ->   00H (FALSE)
```

The space at the end of the second string makes the second string greater than the first one.

```
%GTS(16,111H)    ->   0ffffH (TRUE)
```

ASCII '6' is greater than ASCII '1'.

The arguments can also contain macro calls:

```
%MATCH(NEXT,LIST)(CAT,DOG_MOUSE)

%EQS(%NEXT,CAT)              -> 0ffffH (TRUE)
%EQS(DOG,%SUBSTR(%LIST,1,3))  -> 0ffffH (TRUE)
```

%EVAL

Syntax

%EVAL(expression**)**

Description

The %EVAL function accepts an expression as its argument and returns the expression's value in hexadecimal.

The *expression* argument must be a legal macro-time expression. The return value from %EVAL is built according to macro processing rules for representing hexadecimal numbers. The trailing character is always the hexadecimal suffix (**H**). The expanded value is at most 16 bits and negative numbers are shown in two's complement form. If the leading digit of the return value is 'A', 'B', 'C', 'D', 'E' or 'F', it is preceded by a 0.

Example

```
COUNT SET %EVAL(33H + 15H + 0f00H)      -> COUNT SET 0f48H

MOV  R1, #%EVAL(10H – ((13+6) *2 ) +7)  -> MOV  R1, #0fff1H

%SET( NUM1, 44)   -> null string
%SET( NUM2, 25)   -> null string

MOV  R1, #%EVAL( %NUM1 LE %NUM2 )       -> MOV  R1, #00H
```

%EXIT

Syntax

```
%EXIT
```

Description

The built-in function `%EXIT` terminates expansion of the most recently called user defined macro. It is most commonly used to avoid infinite loops (e.g. a recursive user defined macro that never terminates). It allows several exit points in the same macro.

Example

This example uses the `%EXIT` function to terminate a recursive macro when an odd number of bytes have been added.

```
%*DEFINE (MEM_ADD_MEM(SOURCE,DEST,BYTES))
(     %IF( %BYTES LE 0 ) THEN ( %EXIT ) FI
      ADD   A, %SOURCE
      ADDC  A, %DEST
      MOV   %DEST, A
      %IF( %BYTES EQ 1 ) THEN ( %EXIT ) FI
      MOV   A, %SOURCE+1
      ADDC  A, %DEST+1
      MOV   %DEST+1, A
      %IF(%BYTES GT 2) THEN (
          %MEM_ADD_MEM(%SOURCE+2,%DEST+2,%BYTES-2)) FI
)
```

The above example adds two pairs of bytes and stores results in DEST. As long as there is a pair of bytes to be added, the macro MEM_ADD_MEM is expanded. When BYTES reaches a value of 1 or 0, the macro is exited.

In the following example `%EXIT` is a simple jump out of a recursive loop:

```
%*DEFINE (BODY)
(     MOV A,%MVAR
      %SET(MVAR, %MVAR + 1 )
)

%*DEFINE (UNTIL(CONDITION,EXE_BODY))
(     %EXE_BODY
      %IF( %CONDITION )
      THEN (
         %EXIT )
      ELSE (
         %UNTIL( %CONDITION, %EXE_BODY )
      )FI
)
```

```
%SET(MVAR,0)
%UNTIL( %MVAR GT 3, %*BODY )
```

Related Information

%REPEAT

%WHILE

%__FILE__, %__LINE__

Syntax

```
%__FILE__
%__LINE__
```

Description

The `%__FILE__` macro is equivalent to the ISO C predefined macro, it translates into the name of the current source file.

The `%__LINE__` macro is equivalent to the ISO C predefined macro, it translates into the line number of the current source line.

Example

```
%ERROR(Error in file %__FILE__ on line %__LINE__)
```

%IF

Syntax

```
%IF(expression)
THEN
 (text1)
[ELSE]             ; the ELSE part is optional
 (text2)]
FI
```

Description

With the `%IF` function 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 result of the expression is TRUE, then the succeeding *text1* is expanded; if it is FALSE and the optional `ELSE` clause is included in the call, then the *text2* is expanded. If the expression results FALSE and the `ELSE` clause is not included, the `IF` call returns the null string. The macro call must be terminated by `FI`.

You can nest `%IF` calls to any level. The `ELSE` clause refers to the most recent `%IF` call that is still open (not terminated by `FI`). `FI` terminates the most recent `%IF` call that is still open.

Example

This is a simple example of the IF call with no ELSE clause:

```
%SET( VALUE, 0F0H )
%IF( %VALUE GE 0FFH )
THEN
     ( MOV R1, #%VALUE)
FI
```

This is a simple form of the IF call with an ELSE clause:

```
%IF( %EQS(ADD,%OPERATION))
THEN
     ( ADD R7, #03H)
ELSE
     ( SUB R7, #03H)
FI
```

This is an example of three nested IF calls:

```
%IF( %EQS(%OPER,ADD)) THEN (
     ADD  R1, #03H
)ELSE (%IF( %EQS(%OPER,SUB)) THEN (
```

```
      SUB  R1, #03H
      )ELSE (%IF( %EQS(%OPER,MUL)) THEN (
          MOV  R1, #03
          JMP  MUL_LAB
          )ELSE (
              MOV R1, #DATUM
              JMP DIV_LAB
          )FI
      )FI
)FI
```

Demonstrating conditional assembly:

```
%SET(DEBUG,1)
%IF(%DEBUG)
THEN (
      MOV  R1, #%DEBUG
      JMP  DEBUG
)FI

      MOV  R1, R2
        .
        .
        .
```

This expands to:

```
      MOV  R1, #01H
      JMP  DEBUG
      MOV  R1, R2
```

To turn of the debug code you can change `%SET` to:

```
%SET(DEBUG,0)
```

Related Information

%IFDEF, %IFNDEF

%IFDEF, %IFNDEF

Syntax

```
%IFDEF(macro)
THEN
 (text1)
[ELSE]             ; the ELSE part is optional
 (text2)]
FI

%IFNDEF(macro)
THEN
 (text1)
[ELSE]             ; the ELSE part is optional
 (text2)]
FI
```

Description

The `%IFDEF` built-in function tests if a macro is defined and the `%IFNDEF` built-in function tests if a macro is not defined. Based on this test, the function expands or withholds its text arguments. These functions allow you to decide at macro time whether to assemble certain code or not (conditional assembly). So, the assembler never has to see any code which is not to be assembled.

The `%IFDEF` and `%IFNDEF` functions first test if *macro* is defined (IFDEF) or not (IFNDEF). If it is TRUE, then the succeeding *text1* is expanded; if it is FALSE and the optional `ELSE` clause is included in the call, then the *text2* is expanded. If the test results to FALSE and the `ELSE` clause is not included, the macro call returns the null string. The macro call must be terminated by `FI`.

You can nest `%IFDEF`/`%IFNDEF` calls to any level. The `ELSE` clause refers to the most recent call that is still open (not terminated by `FI`). `FI` terminates the most recent `%IFDEF`/`%IFNDEF` call that is still open.

Example

This is a simple example of the IFNDEF call with no ELSE clause:

```
%IFNDEF(MODEL)
THEN (
%DEFINE(MODEL)(SMALL)
) FI
```

This is a simple form of the IFDEF call with an ELSE clause:

```
%IFDEF(DOADD)
THEN
     ( ADD R7, #03H)
ELSE
     ( SUB R7, #03H)
FI
```

Related Information

`%IF`

%IN, %OUT

Syntax

`%IN`

`%OUT(`*text*`)`

Description

These built-in functions perform console I/O. They are line oriented. `%IN` outputs the character '>' as a prompt to the console (unless you specify another prompt with option **--prompt**), and returns the next line typed at the console including the line terminator. `%OUT` outputs a string to the console; the return value of `%OUT` is the null string.

Example

```
%OUT(ENTER NUMBER OF PROCESSORS IN SYSTEM)
%SET(PROC_COUNT,%IN)
%OUT(ENTER THIS PROCESSOR'S ADDRESS)
%SET(ADDRESS,%IN)
%OUT(ENTER BAUD RATE)
%SET(BAUD,%IN)
```

The following lines would be displayed on the console:

```
ENTER NUMBER OF PROCESSORS IN SYSTEM
> user response
ENTER THIS PROCESSOR'S ADDRESS
> user response
ENTER BAUD RATE
> user response
```

Related Information

`%OPTION`

Assembler option **--prompt**

%INCLUDE

Syntax

%INCLUDE(`filename"` | `<filename>`

Description

With the `%INCLUDE` function you include another file at the exact location where the `%INCLUDE` occurs. This happens at macro preprocessing time, before the resulting file is assembled. The `%INCLUDE` function 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` function. 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. Leading and trailing whitespaces are skipped. 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 directory of the current source file.

2. The path that is specified with the assembler option **--include-directory**.

3. The path that is specified in the environment variable `AS51INC` when the product was installed.

4. The default `include` directory in the installation directory.

The state of the assembler is not changed when an include file is processed. The lines of the include file are inserted just as if they belong to the file where it is included.

Example

It is allowed to start a new section in an included file. If this file is included somewhere in another section, the contents of that section following the included file will belong to the section started in the include file:

```
; file incfile.asm

insect .segment data
       .rseg data
       .db 5
       .db 6

; file mainfile.asm

mainsect .segment data
         .rseg data
         .db 1
         .db 2
         %INCLUDE(incfile.asm)
```

```
        .db 3
        .db 4
```

The resulting sections have the following contents:

```
mainsect: 0x01 0x02
incsect:  0x05 0x06 0x03 0x04
```

%LEN

Syntax

%LEN(_text_**)**

Description

The built-in function `%LEN` takes a character string argument and returns the length of the character string in hexadecimal format (the same format as `%EVAL`).

Example

```
Before Macro Expansion              After Macro Expansion

%LEN(ABCDEFGHIJKLMNOPQRSTUVWXYZ)     -> 1aH
%LEN(A,B,C)                          -> 05H
%LEN()                               -> 00H

%MATCH(STR1,STR2) (Cheese,Mouse)
%LEN(%STR1)                          -> 06H
%LEN(%SUBSTR(%STR2, 1, 3 ))          -> 03H
```

Related Information

%MATCH

%SUBSTR

%MATCH

Syntax

```
%MATCH(macro_id1 delimiter macro_id2) (text)
```

Description

The built-in function `%MATCH` primarily serves to define macro identifiers. The `%MATCH` function searches a character string for a delimiter character and assigns the substrings on either side of the delimiter to the macro identifiers.

delimiter is the first character to follow *macro_id1*. You can use a space or a comma or any other delimiter. See Section 2.8.4, *Macro Delimiters* for more information on delimiters.

`%MATCH` searches the *text* for the first *delimiter*. When it is found, all characters to the left of it are assigned to *macro_id1* and all characters to the right are assigned to *macro_id2*. If the delimiter is not found, the entire *text* is assigned to *macro_id1* and the null string is assigned to *macro_id2*.

Example

```
%MATCH(MS1,MS2) (ABC,DEF)        -> MS1=ABC   MS2=DEF
%MATCH(MS3,MS4) (GH,%MS1)        -> MS3=GH    MS4=ABC
%MATCH(MS5,MS6) (%LEN(%MS1))     -> MS5=03H   MS6=null
```

You can use the MATCH function for processing string lists as shown in the next example.

```
%MATCH(NEXT,LIST)(10H,20H,30H)
%WHILE(%LEN(%NEXT))
(    MOV  A, %NEXT
     ADD  A, #2
     MOV  %NEXT, A
     %MATCH(NEXT,LIST)(%LIST)
)
```

Produces the following code:

First iteration of WHILE:

```
     MOV  A, 10H
     ADD  A, #2
     MOV  10H, A
```

Second iteration of WHILE:

```
     MOV  A, 20H
     ADD  A, #2
     MOV  20H, A
```

Third iteration of WHILE:

```
MOV  A, 30H
ADD  A, #2
MOV  30H, A
```

Related Information

%LEN

%SUBSTR

%METACHAR

Syntax

%METACHAR(*text***)**

Default: **%**

Description

You can use this function to redefine the metacharacter (initially: '**%**').

Although the *text* string may be any number of characters long, only the **first** character in the string is taken to be the new metacharacter. Macro calls in the text string are still recognized and corresponding actions that will not lead to any direct expansion on the output file will be performed. So, for example a %SET macro call inside the text string will be performed.

Characters that may not be used as a metacharacter are: a blank, letter, digit, left or right parenthesis, or asterisk.

Example

The following example is catastrophic !!!

```
%METACHAR( & )
```

This examples defines the space character as the new metacharacter, since it is the first character in the text string!

The correct way should be:

```
%METACHAR(&)
```

%OPTION

Syntax

%OPTION(*command_line_option***)**

Description

You can use the `%OPTION` function to trigger a command line option from within the source file.

The *command_line_option* must be any valid command line option. The `%OPTION` function itself is replaced with the null string.

Example

The following command sets the prompt for the `%IN` function to "`y/n: `" from with the source:

```
%OPTION(--prompt=y/n: )
```

Related Information

`%IN`

%REPEAT

Syntax

%REPEAT(*expression***)**
 (*text***)**

Description

Unlike the `%IF` and `%WHILE` macros, `%REPEAT` uses the *expression* for a numerical value that specifies the number of times the *text* should be expanded. The *expression* is evaluated once when the macro is first called, then the specified number of iterations is performed.

A call to built-in function `%EXIT` always terminates a `%REPEAT` macro.

Example

```
Lab:
     MOV A, #8
     MOV R2, #0FFFFH

     %REPEAT( 8 )
     (   MOV @R2, A
         ADD @R1, A
     )
```

Related Information

%EXIT

%WHILE

%SET

Syntax

%SET(_macro_variable_**,**_expression_**)**

Description

The %SET function assigns the value of the numeric _expression_ to the identifier, _macro_variable_, and stores the _macro_variable_ in the macro time symbol table. _macro_variable_ must follow the same syntax convention used for other macro identifiers. Expansion of a _macro_variable_ always results in hexadecimal format.

The %SET macro call affects the macro time symbol table only; when %SET is encountered, the macro preprocessor replaces it with the null string. Symbols defined by %SET can be redefined by a second %SET call, or defined as a macro by a %DEFINE call.

Example

```
%SET(COUNT,0)                 -> null string
%SET(OFFSET,16)               -> null string
MOV R1, #%COUNT + %OFFSET     -> MOV R1,#00H + 10H
MOV R2, #%COUNT               -> MOV R2,#00H
```

%SET can also be used to redefine symbols in the macro time table:

```
%SET(COUNT,%COUNT + %OFFSET)  -> null string
%SET(OFFSET,%OFFSET * 2)      -> null string
MOV R1, #%COUNT + %OFFSET     -> MOV R1,#10H + 20H
MOV R2, #%COUNT               -> MOV R2,#10H
```

%SUBSTR

Syntax

%SUBSTR(_string,start,count_**)**

Description

The built-in function %SUBSTR returns a substring of its text argument. The macro takes three arguments: a string from which the substring is to be extracted and two numeric arguments.

start specifies the starting character of the substring.

count specifies the number of characters to be included in the substring.

If _start_ is zero or greater than the length of the argument string, %SUBSTR returns the null string.

If _count_ is zero, then %SUBSTR returns the null string. If it is greater than the remaining length of the string, then all characters from the start character of the substring to the end of the string are included.

Example

```
Before Macro Expansion          After Macro Expansion


%SUBSTR(ABCDEFG, 5, 1 )         -> E
%SUBSTR(ABCDEFG, 5, 100 )       -> EFG
%SUBSTR(123(56)890, 4, 4 )      -> (56)
%SUBSTR(ABCDEFG, 8, 1 )         -> null
%SUBSTR(ABCDEFG, 3, 0 )         -> null
```

Related Information

%LEN

%MATCH

%UNDEF

Syntax

%UNDEF(_identifier_**)**

Description

You can use this function to undefine a previously defined macro, or one of the built-in macro functions.

The _identifier_ must be a previously defined macro name or one of the built-in functions. The %UNDEF command is replaced with the null string.

Example

```
%DEFINE(TEMP)(path)    -> macro TEMP is defined

%UNDEF(TEMP)           -> null string, TEMP is undefined
%UNDEF(SET)            -> null string
%SET(COUNT,0)          -> undefined macro name: SET
```

%WHILE

Syntax

```
%WHILE(expression)
 (text)
```

Description

The `%WHILE` built-in function evaluates the *expression*. If it results to TRUE, the *text* is expanded. `%WHILE` expands to the null string. Once the *text* has been expanded, the logical argument is retested and if it is still TRUE, the *text* is expanded again. This continues until the logical argument proves FALSE.

Since the macro continues processing until the *expression* is FALSE, the *text* should modify the *expression*, or else `%WHILE` may never terminate.

A call to built-in function `%EXIT` always terminates a `%WHILE` macro.

Example

This example uses the `%SET` macro and a macro-time symbol to count the iterations of the `%WHILE` macro.

```
%SET(COUNTER,7)

%WHILE( %COUNTER GT 0 )
(    MOV   R2, #%COUNTER
     MOV   @R1, R2
     ADD   R1, #2
     %SET(COUNTER, %COUNTER - 1)
)
```

Related Information

%EXIT

%REPEAT

2.8.4. Macro Delimiters

Delimiters are used in the function `%DEFINE` to separate the macro name from the optional parameter list and to separate different parameters in this parameter list. In the `%MATCH` function a delimiter is used to define a separator, which is used as kind of terminator in the corresponding balanced text argument. The most commonly used delimiters are characters like parentheses and commas, but the macro language permits almost any character or group of characters to be used as a delimiter.

Regardless of the type of delimiter used to define a macro, once it has been defined, only the delimiters used in the definition can be used in the macro call. Macros defined with parentheses and commas require parentheses and commas in the macro call. Macros defined with spaces (or any other delimiter), require that delimiter when called.

Macro delimiters can be divided into three classes: implied blank delimiters, identifier delimiters, and literal delimiters.

2.8.4.1. Implied Blank Delimiters

Implied blank delimiters are the easiest to use and contribute the most readability and flexibility to macro definitions. An implied blank delimiter is one or more spaces, tabs or new lines (a carriage-return/linefeed pair) in any order. To define a macro that uses the implied blank delimiter, simply place one or more spaces, tabs, or new lines surrounding the parameter list and separating the formal parameters.

When you call the macro defined with the implied blank delimiter, each delimiter will match a series of spaces, tabs, or new lines. Each parameter in the call begins with the first non-blank character, and ends when a blank character is found.

Example

```
%*DEFINE(WORDS FIRST SECOND)(TEXT: %FIRST %SECOND)
```

All of the following calls are valid:

```
Before Macro Expansion      After Macro Expansion


%WORDS hello world          -> TEXT: hello world
%WORDS    one
          two               -> TEXT: one two
%WORDS
          well
done                        -> TEXT: well done
```

2.8.4.2. Identifier Delimiters

Identifier delimiters are legal macro identifiers designated as delimiters. To define a macro that uses an identifier delimiter in its call pattern, you must prefix the delimiter with the commercial at symbol '@'. You must separate the identifier delimiter from the macro identifiers by a blank character.

When calling a macro defined with identifier delimiters, an implied blank delimiter is required to precede the identifier delimiter, but none is required to follow the identifier delimiter.

Example

```
%*DEFINE(ADD M1 @TO M2 @AND M3)(
     MOV A,%M1
     ADD A,%M2
     MOV %M2,A
     MOV A,%M1
     ADD A,%M3
     MOV %M3,A
)
```

The following call (there is no blank after TO and AND):

```
%ADD ATOM TOBILL ANDLIST
```

returns the following code after expansion:

```
MOV A,ATOM
ADD A,BILL
MOV BILL,A
MOV A,ATOM
ADD A,LIST
MOV LIST,A
```

2.8.4.3. Literal Delimiters

The delimiters we used with the user-defined macros (parentheses and commas) were literal delimiters. A literal delimiter can be any character except the metacharacter.

When you define a macro using a literal delimiter, you must use exactly that delimiter when you call the macro.

When defining a macro, you must literalize the delimiter string, if the delimiter you wish to use meets any of the following conditions:

• uses more than one character

• uses a macro identifier character (A-Z, , _, or ?)

• uses a commercial at (@)

• uses a space, tab, carriage-return, or linefeed

You can use the escape function (%*n*) or the bracket function (%()) to literalize the delimiter string.

Example

```
Before Macro Expansion              After Macro Expansion

%*DEFINE(MAC(A,B))(%A  %B)          -> null string
%MAC(2,3)                           -> 2  3
```

In the following example brackets are used instead of parentheses. The commercial at symbol separates the parameters:

```
%*DEFINE(OR[A%(@)B])(OR %A,%B)      -> null string
%OR[A1@A2]                          -> OR A1,A2
```

In the next example, delimiters that could be identifier delimiters have been defined as literal delimiters:

```
%*DEFINE(ADD(A%(AND)B))(AND %A,%B)  -> null string
%ADD (A AND #34H)                   -> AND A , #27H
```

The spaces around AND are considered as part of the argument string.

Example

The next example demonstrates the difference between identifier delimiters and literal delimiters.

```
%*DEFINE(ADD M1%(TO)M2%(AND)M3)(
     MOV A,%M1
     ADD A,%M2
     MOV %M2,A
     MOV A,%M1
     ADD A,%M3
     MOV %M3,A
)
```

The following call:

```
%ADD ATOM TOBILL ANDLIST
```

returns the following code after expansion (the `TO` in `ATOM` is recognized as the delimiter):

```
     MOV A,A
     ADD A,M TOBILL
     MOV M TOBILL,A
     MOV A,A
     ADD A,LIST
     MOV LIST,A
```

2.8.5. Literal Mode versus Normal Mode

In *normal mode*, the macro preprocessor scans text looking for the metacharacter. When it finds one, it begins expanding the macro call. Parameters and macro calls are expanded. This is the usual operation of the macro preprocessor, but sometimes it is necessary to modify this mode of operation. The most common use of the *literal mode* is to prevent macro expansion. The literal character in DEFINE prevents the expansion of macros in the macro body until you call the macro.

When you place the literal character '*****' in a DEFINE call, the macro preprocessor shifts to literal mode while expanding the call. The effect is similar to surrounding the entire call with the bracket function. Parameters to the literalized call are expanded, the escape, comment, and bracket functions are also expanded, but no further processing is performed. If there are any calls to other, they are not expanded.

If there are no parameters in the macro being defined, the DEFINE built-in function can be called without the literal character. If the macro uses parameters, the macro will attempt to evaluate the formal parameters in the macro body as parameterless macro calls.

Example

The following example illustrates the difference between defining a macro in literal mode and normal mode:

```
%SET(TOM,1)

%*DEFINE (M1)(
     %EVAL(%TOM)
)

%DEFINE (M2)(
```

```
    %EVAL(%TOM)
)
```

When `M1` and `M2` are defined, `TOM` is equal to 1. The macro-body of `M1` has not been evaluated due to the literal character, but the macro body of `M2` has been completely evaluated, since the literal character is not used in the definition. Changing the value of `TOM` has no affect on `M2`, it changes the return value of `M1` as illustrated below:

```
Before Macro Expansion    After Macro Expansion


%SET(TOM,2)
%M1                       -> 02H
%M2                       -> 01H
```

The macros themselves can be called with the literal character. The return value then is the unexpanded body:

```
%*M2                      -> 01H
%*M1                      -> %EVAL(%TOM)
```

Example

Sometimes it is necessary to obtain access to parameters by several macro levels. The literal mode is also used for this purpose. The following example assumes that the macro `M1` called in the macro-body is predefined.

```
@*DEFINE (M2(P1))(
    MOV  A,%P1
    %M1(%P1)
)
```

In the above example, the formal parameter `%P1` is used once as a simple place holder and once as an actual parameter for the macro `M1`.

Actual parameters in the contents must not be known in literal mode, since they are not expanded. If the definition of `M2`, however, occurred in normal mode, the macro preprocessor would try to expand the call from `M1` and, therefore, the formal parameter `%P1` (used as an actual parameter). However, this first receives its value when called from `M2`. If its contents happen to be undefined, an error message is issued.

Example

Another application possibility for the literal mode exists for macro calls that are used as actual parameters (macro strings, macro variables, macro calls).

```
%M1(%*M2)
```

The formal parameter of `M1` was assigned the call from `M2` ('%M2') by its expansion. `M2` is expanded from `M1` when the formal parameters are processed.

In normal mode, `M2` is expanded in its actual parameter list immediately when called from `M1`. The formal parameters of `M1` in its body are replaced by the prior expanded macro body from `M2`.

Example

The following example shows the different use of macros as actual parameters in the literal and normal mode.

```
%SET(M2,1)

%*DEFINE (M1(P1))(
     %SET(M2,%M2 + 1)
     %M2, %P1
)

%M1(%*M2)                 -> 02H, 02H
%M1(%M2)                  -> 03H, 02H
%M1(%*M2)                 -> 04H, 04H
```

2.8.6. Algorithm for Evaluating Macro Calls

The algorithm of the macro preprocessor used for evaluating the source file can be broken down into 6 steps:

1. Scan the input stream until the metacharacter is found.

2. Isolate the macro name.

3. If macro has parameters, expand each parameter from left to right (initiate step one on actual parameter), before expanding the next parameter.

4. Substitute actual parameters for formal parameters in macro body.

5. If the literal character is not used, initiate step one on macro body.

6. Insert the result into output stream.

The terms 'input stream' and 'output stream' are used because the return value of one macro may be a parameter to another. On the first iteration, the input stream is the source line. On the final iteration, the output stream is passed to the assembler.

Example

The examples below illustrate the macro preprocessor's evaluation algorithm:

```
%SET(TOM,3)

%*DEFINE (STEVE)(%SET(TOM,%TOM - 1) %TOM)

%DEFINE (ADAM(A,B))(
     DB  %A, %B, %A, %B, %A, %B
)
```

The call ADAM is presented here in the normal mode with TOM as the first actual parameter and STEVE as the second actual parameter. The first parameter is completely expanded before the second parameter is expanded. After the call to ADAM has been completely expanded, TOM will have the value 02H.

Before Macro Expansion   After Macro Expansion

```
%ADAM(%TOM,%STEVE)      -> DB 03H, 02H, 03H, 02H, 03H, 02H
```

Now reverse the order of the two actual parameters. In this call to ADAM, STEVE is expanded first (and TOM is decremented) before the second parameter is evaluated. Both parameters have the same value.

```
%SET(TOM,3)
%ADAM(%STEVE,%TOM)      -> DB 02H, 02H, 02H, 02H, 02H, 02H
```

Now we will literalize the call to STEVE when it appears as the first actual parameter. This prevents STEVE from being expanded until it is inserted in the macro-body, then it is expanded for each replacement of the formal parameters. TOM is evaluated before the substitution in the macro body.

```
%SET(TOM,3)
%ADAM(%*STEVE,%TOM)     -> DB 02H, 03H, 01H, 03H, 00H, 03H
```

2.9. 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 directives

  - Data definition / Storage allocation directives

  - High Level Language (HLL) directives

- 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 and $NOLIST you overrule this option for a part of the code that you do not want to appear in the list file. Controls always appear on a separate line and start with a '$' 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. The assembler recognizes both uppercase and lowercase for directives and controls.

2.9.1. Assembler Directives

Overview of assembly control directives

Directive	Description
.END	Indicates the end of an assembly module

Overview of symbol definition directives

Directive	Description
.ALIAS	Create an alias for a symbol
.BIT	Assign bit address to a symbol
.CODE	Assign CODE address to a symbol
.DATA	Assign DATA address to a symbol
.EQU	Set permanent value to a symbol
.EXTRN	Import global section symbol
.IDATA	Assign IDATA address to a symbol
.NAME	Define module name
.PUBLIC	Declare global section symbol
.RSEG	Select a section
.SEGMENT	Declare a section
.SET	Set temporary value to a symbol
.WEAK	Mark a symbol as 'weak'
.XDATA	Assign XDATA address to a symbol

Overview of data definition / storage allocation directives

Directive	Description
.DBIT	Define bit
.DB	Define byte
.DW	Define word (16 bits)
.DL	Define long (32 bits)
.DS	Define storage

Overview of register bank directives

Directive	Description
.USING	Use register bank number

Overview of HLL directives

Directive	Description
.CALLS	Pass call tree information and/or stack usage information
.MISRAC	Pass MISRA C information

Overview of directives supported for backwards compatibility

The following directives are not described in this manual, they are only supported by the assembler for backwards compatibility reasons.

Directive	Description
.BSEG	Select absolute BIT section
.CSEG	Select absolute CODE section
.DSEG	Select absolute DATA section
.ISEG	Select absolute IDATA section
.ORG	Modify location counter
.XSEG	Select absolute XDATA section

.ALIAS

Syntax

```
alias-name .ALIAS function-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`.

The *alias-name* cannot be redefined anywhere else in the program (or section, if section directives are being used). Symbols defined with the `.ALIAS` directive can be made public with the `.PUBLIC` directive. The symbol defined with the `.ALIAS` gets the same type as the originating symbol.

Example

```
_malloc .ALIAS malloc
```

.BIT

Syntax

symbol **.BIT** *expression*

Description

With the .BIT directive you assign a BIT address to a *symbol* name. The *expression* must evaluate into a number or BIT address and may not contain forward references. The symbol will be of type BIT.

Example

```
      .RSEG  A_SEG        ;relocatable bit
                          ;addressable section
CTRL: .DS    1

TST   .BIT   CTRL.0       ;bit in relocatable byte
OK    .BIT   TST+1        ;next bit
TST2  .BIT   64H          ;absolute bit
```

Related Information

Section 2.7.4, *Symbol Types and Expression Types*

.EQU (Set permanent value to a symbol)

.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.

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.

Example

The function _main calls the function _nfunc:

.CALLS '_main', '_nfunc'

The function _main() uses 4 bytes on the stack:

.CALLS '_main','',4

.CODE

Syntax

symbol **.CODE** *expression*

Description

With the `.CODE` directive you assign a CODE address to a *symbol* name. The *expression* must evaluate into a number or CODE address and may not contain forward references. The symbol will be of type CODE.

Example

```
RESTART     .CODE     00H
```

Related Information

Section 2.7.4, *Symbol Types and Expression Types*

`.EQU` (Set permanent value to a symbol)

.DATA

Syntax

```
symbol .DATA expression
```

Description

With the .DATA directive you assign a DATA address to a *symbol* name. The *expression* must evaluate into a number or DATA address and may not contain forward references. The symbol will be of type DATA.

Example

```
TSTART    .DATA 60H  ;define TSTART to be at
                     ;location 60H
TEND      .DATA 6DH  ;define TEND to be at
                     ;location 6DH
```

Related Information

Section 2.7.4, *Symbol Types and Expression Types*

.EQU (Set permanent value to a symbol)

.DBIT, .DB, .DW, .DL

Syntax

```
[label] .DBIT argument[,argument]...
[label] .DB argument[,argument]...
[label] .DW argument[,argument]...
[label] .DL argument[,argument]...
```

Description

With these directive you can define memory. With each directive the assembler allocates and initializes one or more bytes 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 must be separated by commas with no intervening spaces. Empty arguments are stored as 0 (zero). For single bit initialization (.DBIT) the argument must be a positive absolute expression and each argument represents a bit to be initialized.

Multiple arguments are stored in successive byte locations. 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 following table shows the number of bits initialized.

Directive	Bits
.DBIT	1
.DB	8
.DW	16
.DL	32

When these directives are used in a BIT section, each argument initializes the number of bits defined for the used directive and the location counter of the current section is incremented with this number of bits.

The .DBIT directive can be used in a BIT section only. Each argument represents a bit to be initialized to 0 or 1. The location counter of the current section is incremented by a number of bits equal to the number of arguments.

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 / long, the assembler issues a warning and truncates the value.

String constants

Single-character strings are stored in a byte whose lower seven bits represent the ASCII value of the character, for example:

```
.DB 'R'          ; = 0x52
```

Multiple-character strings are stored in consecutive byte addresses, as shown below. The standard C language escape characters like '\n' are permitted.

```
.DB 'AB',,'C'  ; = 0x41420043 (second argument is empty)
```

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:

```
WTBL: .DW 'ABC',,'D'  ; results in 0x414200000044 , the 'C' is truncated
LTBL: .DL 'ABC'       ; results in 0x00414243
```

Related Information

.DS (Define Storage)

.DS

Syntax

```
[label] .DS expression
```

Description

The .DS 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). In a bit section, the MAU size is 1, thus the .DS directive will initializes a number of bits equal to the result of the expression.

Example

```
DSEC   .segment data clear
       .rseg data clear
RES:   .DS 5+3   ; allocate 8 bytes
```

Related Information

.DB (Define Memory)

.END

Syntax

.END

Description

With the .END directive you tell the assembler that the end of the module is reached. The assembler will not process any lines following an .END directive.

The assembler does not allow a label with this directive.

Example

```
CSEC .segment code
     .rseg code
       ; source lines
     .END                 ; End of assembly module
```

.EQU

Syntax

symbol **.EQU** *expression*

Description

With the .EQU directive you assign the value of *expression* to *symbol* permanently. The expression can be relative or absolute. Once defined, you cannot redefine the symbol. With the .PUBLIC directive you can declare the symbol global.

The symbol defined with the .EQU gets a type depending on the resulting type of the expression. If the resulting type of the expression is none the symbol gets no type when the .EQU is used outside a section and it gets the type of the section when it is defined inside a section.

Example

To assign the value 0x4000 permanently to the symbol MYSYMBOL:

MYSYMBOL .EQU 0x4000

Related Information

Section 2.7.4, *Symbol Types and Expression Types*

.SET (Set temporary value to a symbol)

.EXTRN

Syntax

.EXTRN *type* **(***symbol*[,*symbol*]...**)** [**,***type* **(***symbol*[,*symbol*]...**)**]...

Description

With the .EXTRN 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 .PUBLIC directive.

The *type* of the *symbol* can be one of the section types BIT, CODE, DATA, IDATA or XDATA or NUMBER. The type NUMBER does not correspond to a specific memory space, but indicates a typeless number. The assembler uses the types to check the symbol's use. In other words, if the symbol does not fit the instruction's operand, the assembler will issue a warning.

If you do not use the .EXTRN directive and the symbol is not defined within the current module, the assembler issues a warning and inserts the .EXTRN directive.

A label is not allowed with this directive.

Example

```
      .EXTRN CODE (AVAR,get_info), DATA(count) ; extern declaration
      .EXTRN BIT(mybit,abit), NUMBER(tnum)

CSEC .segment code
      .rseg code
      .
      .
      MOV R0, AVAR      ; AVAR is used here
      .
```

Related Information

See Section 2.7.4, *Symbol Types and Expression Types* for more information on the *type* keywords.

.PUBLIC (Declare global section symbol)

.IDATA

Syntax

symbol **.IDATA** *expression*

Description

With the `.IDATA` directive you assign a IDATA address to a *symbol* name. The *expression* must evaluate into a number or IDATA address and may not contain forward references. The symbol will be of type IDATA.

Example

```
TSTART    .IDATA 60H  ;define TSTART to be at
                      ;location 60H
TEND      .IDATA 6DH  ;define TEND to be at
                      ;location 6DH
```

Related Information

Section 2.7.4, *Symbol Types and Expression Types*

`.EQU` (Set permanent value to a symbol)

#line

Syntax

#[**line**] *linenumber* ["*filename*"]

Description

The line directive is the only directive not starting with a dot, but with a hash sign. It allows passing on line number information from higher level sources. This *linenumber* is used when generating errors. When this directive is encountered, the internal line number count is reset to the specified number and counting continues after the directive. The line after the directive is assumed to originate on the specified line number. The optional file name will, when specified, reset the module file name for purposes of error generation.

This directive is generated by the preprocessor phase of the C compiler. Normally you will not use it in hand-coded assembly.

Example

```
#line 1
```

.NAME

Syntax

.NAME *string*

Description

With the .NAME directive you can identify the current program module. If this directive is not present, the module name is taken from the input source file name.

Example

```
.NAME my_prog  ; module name is my_prog
```

.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 3.7.2, *C Code Checking: MISRA C*

C compiler option **--misrac**

.PUBLIC

Syntax

.PUBLIC *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 .PUBLIC 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 .PUBLIC, from another module, use the .EXTRN directive.

Only program labels and symbols defined with .EQU can be made global.

The assembler does not allow a label with this directive. The type of the global symbol is determined by its definition.

Example

```
LOOPA .EQU 1          ; definition of symbol LOOPA
      .PUBLIC  LOOPA  ; LOOPA will be globally
                      ; accessible by other modules
```

Related Information

.EXTRN (Import global section symbol)

.RSEG

Syntax

.RSEG *name type* [*attribute*...]

Description

Use this directive to switch to a section previously defined by a .SEGMENT directive. The following statements will be assembled in the section *name*, using the location counter of the named section. The specified section remains in effect until another .SEGMENT or .RSEG directive is encountered. The location counter of the section is initially set to zero.

The *type* and *attributes* are the same as that of the .SEGMENT directive.

Example

```
CSEC   .SEGMENT code
       .RSEG CSEC code    ;select code section

ABSSEC .SEGMENT xdata at(0x100)
       .RSEG ABSSEC xdata at(0x100)
    ; absolute section

     .RSEG CSEC code     ;switch to CSEC again
```

Related Information

Section 2.7.4, *Symbol Types and Expression Types*.

.SEGMENT (Declare section)

.SEGMENT

Syntax

name **.SEGMENT** *type* [*attribute*...]

Description

Use this directive to declare a section, assign a set of section attributes, and initialize the location counter to zero.

The *name* specifies the name of the section. The *type* operand specifies the section's space and must be one of:

Section type	Description
BIT	bit address space
CODE	code address space
DATA	direct addressable data
IDATA	indirect addressable data
XDATA	external data space

The section type and attributes are case insensitive.

The defined *attribute*s are:

Attribute	Description
AT (*address*)	Locate the section at the given *address*.
BITADDRESSABLE	Specifies a section to be relocated within the bit space on a byte boundary (BDATA). Allowed only with DATA sections and the section size is limited to 16 bytes.
CLEAR	Sections are zeroed at startup.
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). To prevent naming conflicts with other symbols, the prefix ".cluster." is added to the cluster name during object file generation.
COMMON	Specifies a section to be located in the common area. Allowed only with CODE segments. This is only useful when code bank switching is used.
INBLOCK	Specifies a section which must be contained in a 2048-byte page. Allowed only with CODE sections.
INIT	Defines that the section contains initialization data, which is copied from ROM to RAM at program startup.
INPAGE	Specifies a section which must be contained in a 256-byte page. Allowed only with CODE and XDATA sections.
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

Attribute	Description
OVERLAY (*b*[,*b*]...)	Specifies the register banks (*b*) used in the section. This information will be used by the linker to overlay sections using the same register banks. No overlaying will be done when the OVERLAY attribute is omitted. The OVERLAY attribute is not allowed with CODE sections.
PAGE	Specifies a section which start address must be on a 256-byte page boundary. Allowed only with CODE and XDATA sections.
PROTECT	Tells the linker to exclude a section from unreferenced section removal and duplicate section removal.
ROMDATA	Specifies that the section contains initialized data. This attribute is allowed with CODE and XDATA sections only. This information is meaningful to allocate constant data in the XDATA memory space. When used with CODE sections it is only meaningful for debugging purposes. A section that has been declared with the ROMDATA attribute cannot be disassembled by a debugger.
SHORT	XDATA sections can be declared with the SHORT attribute. The linker allocates the section in a page of auxiliary memory (PDATA).
UNIT	The default relocation attribute: the section will not be aligned.

Example

```
DSEC   .SEGMENT data init
       .RSEG DSEC data init
TAB2   .DW 8   ; initialized section

ABSSEC .SEGMENT xdata at(0x100)
       .RSEG ABSSEC xdata at(0x100)
     ; absolute section
```

Related Information

Section 2.7.4, *Symbol Types and Expression Types*.

`.RSEG` (Select section)

.SET

Syntax

symbol **.SET** *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 `.PUBLIC` directive.

The `.SET` directive is useful in establishing temporary or reusable counters within macros. *expression* must be absolute and cannot include a symbol that is not yet defined (no 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)

.USING

Syntax

.USING *expression*

Description

With the .USING directive you specify the register bank that is used by the subsequent code. The expression is the number (between 0 and 3 inclusive) which refers to one of the four register banks.

The .USING directive allows you to use the predefined symbolic register addresses (AR0 through AR7) instead of their absolute addresses. In addition, the directive causes the assembler to reserve a space for the specified register bank.

Note that if you equate a symbol (e.g. with a .EQU directive) to an AR*i* symbol, the user-defined symbol will not change its value as a result of the subsequent .USING directive.

Example

```
    .USING  3
    PUSH   AR2  ;Push register 2 of bank 3

    .USING  1
    PUSH   AR2  ;Push register 2 of bank 1
```

Related Information

.EQU (Set permanent value to a symbol)

.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 .PUBLIC directive or the .EXTRN 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 .PUBLIC 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
      .PUBLIC  LOOPA  ; LOOPA will be globally
                      ; accessible by other modules
      .WEAK LOOPA     ; mark symbol LOOPA as weak
```

Related Information

.EXTRN (Import global section symbol)

.PUBLIC (Declare global section symbol)

.XDATA

Syntax

symbol **.XDATA** *expression*

Description

With the .XDATA directive you assign a XDATA address to a *symbol* name. The *expression* must evaluate into a number or XDATA address and may not contain forward references. The symbol will be of type XDATA.

Example

```
        .RSEG   XSPACE

ROOM:   .DS     4       ;reserve 4 bytes of XDATA
MORE_X  .XDATA  ROOM+2  ;define MORE_X to be 2
                        ;bytes after ROOM
```

Related Information

Section 2.7.4, *Symbol Types and Expression Types*

.EQU (Set permanent value to a symbol)

2.9.2. Assembler Controls

Assembler controls are classified as *primary* or *general*.

Primary controls affect the overall behavior of the assembler and remain in effect throughout the assembly. For this reason, primary controls may only be used on the command line (with option **--control**) or at the beginning of a source file, before the assembly starts. If you specify a primary control more than once, a warning message is given and the last definition is used. This enables you to override primary controls via the invocation line (with **--control**).

General controls are used to control the assembler during assembly. Control lines containing general controls may appear anywhere in a source file and are also allowed on the command line (with **--control**). When you specify general controls via the command line the corresponding general controls in the source file are ignored.

Controls start with a **$** as the first character on the line. Unknown controls are ignored after a warning is issued. The arguments of controls can optionally be enclosed in braces (). All controls have abbreviations of 2 characters (or 4 characters for the $no.. variant).

Overview of assembler controls

Control	Class	Description
`$[NO]ASMLINEINFO`	General	Generate source line information for assembly files
`$[NO]CASE`	Primary	User-defined symbols are case (in)sensitive
`$DATE`	Primary	Set the date in the list file page header
`$[NO]DEBUG`	Primary	Control debug information generation
`$EJECT`	General	Generate form feed in list file page header
`$[NO]ERRORPRINT`	Primary	Print errors to a file
`$[NO]LIST`	General	Print source lines to list file
`$MESSAGE`	General	Programmer generated message
`$[NO]MOD51`	General	Use predefined register names
`$NOEXTERNALMEMORY`	General	Assemble for derivatives without external memory
`$[NO]OBJECT`	Primary	Alternative name for object file
`$[NO]OPTIMIZE`	General	Control optimization
`$PAGELENGTH`	Primary	Set list file page length
`$PAGEWIDTH`	Primary	Set list file page width
`$[NO]PAGING`	Primary	Control pagination of list file
`$[NO]PRINT`	Primary	Generate a list file
`$[NO]REGADDR`	General	Allow/disallow operands to refer to an absolute register address
`$[NO]REGISTERBANK`	Primary	Specify register banks used
`$SAVE / $RESTORE`	General	Save and restore the current value of the `$LIST / $NOLIST` controls

Control	Class	Description
$SMALLROM	Primary	Application fits in one 2K byte block
$TITLE	General	Set program title in header of assembly list file

$ASMLINEINFO / $NOASMLINEINFO

Syntax

```
$ASMLINEINFO
$NOASMLINEINFO
```

Default

```
$NOASMLINEINFO
```

Abbreviation

```
$AL / $NOAL
```

Description

With the $ASMLINEINFO control the assembler generates assembly level debug information. This matches the effect of the **--debug-info=+asm** (**-ga**) command line option. When you use the command line option, it sets the default, but the control will override its effect.

Example

```
$ASMLINEINFO
    ;generate line and file debug information
    MOV R0, R1
$NOASMLINEINFO
    ;stop generating line and file information
```

Related Information

Assembler option **--debug-info**

Assembler control **$DEBUG**

$DATE

Syntax

$DATE(*string***)**

Abbreviation

$DA

Description

This control sets the date as subtitle of the list file page header. When no $DATE is used the assembler uses the date and time when the list file was generated. The string argument of the $DATE control is not checked for a valid date, in fact any string can be used.

Example

```
; Jul 28 2017 in header of list file
$date('Jul 28 2017')
```

Related Information

Assembler option **--list-file**

$CASE / $NOCASE

Syntax

```
$CASE
$NOCASE
```

Default

```
$CASE
```

Abbreviation

```
$CA / $NOCA
```

Description

Selects whether the assembler operates in case sensitive mode or not. In case insensitive mode the assembler maps characters on input to uppercase (literal strings excluded).

Related Information

Assembler option **--case-insensitive**

$DEBUG / $NODEBUG

Syntax

```
$DEBUG
$NODEBUG
```

Default

```
$NODEBUG
```

Abbreviation

```
$DB / $NODB
```

Description

With the $DEBUG control you enable the assembler to generate debug information. If no high-level language debug information is present, debug information on assembly level is generated. This control also generates debug information on local symbols. This matches the effect of the **--debug-info=+local,+smart** (**-gls**) command line option. When you use the command line option, it sets the default, but the control will override its effect.

Example

```
$DEBUG
   ;generate smart debug information and information on local symbols
   MOV R0, R1
```

Related Information

Assembler option **--debug-info**

Assembler control **$ASMLINEINFO**

$EJECT

Syntax

`$EJECT`

Default

A new page is started when the page length is reached.

Abbreviation

`$EJ`

Description

If you generate a list file with the assembler option **--list-file**, with the $EJECT control the list file generation advances to a new page by inserting a form feed. The new page is started with a new page header. The $EJECT control has no effect when $NOPAGING is set.

Example

```
    .       ; assembler source lines
    .
$EJECT      ; generate a formfeed
    .
```

Related Information

Assembler option **--list-file**

Assembler control **$PAGING**

$ERRORPRINT / $NOERRORPRINT

Syntax

$ERRORPRINT(*file***)**
$NOERRORPRINT

Default

$NOERRORPRINT

Abbreviation

$EP / $NOEP

Description

With the $ERRORPRINT control you can redirect the error messages, normally displayed at the console, to an error list *file*.

Example

```
$ep(errlist.ers)  ; redirect errors to file errlist.ers
```

$LIST / $NOLIST

Syntax

```
$LIST
$NOLIST
```

Default

```
$LIST
```

Abbreviation

```
$LI / $NOLI
```

Description

If you generate a list file with the assembler option **--list-file**, you can use the $LIST/$NOLIST 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.

Example

```
    ...  ; source line in list file
$NOLIST
    ...  ; source line not in list file
$LIST
    ...  ; source line also in list file
```

Related Information

Assembler option **--list-file**

Assembler control **$SAVE / $RESTORE**

$MESSAGE

Syntax

$MESSAGE(*string* **)**

Abbreviation

$MSG

Description

With the $MESSAGE control you tell the assembler to print a message to stderr during the assembling process. This is for example useful in combination with conditional assembly to indicate which part is assembled.

Example

```
    %DEFINE(ID)(4)
$MESSAGE(The value of ID is %ID)
```

The assembler prints the following message to stderr:

```
The value of ID is 4
```

$MOD51 / $NOMOD51

Syntax

```
$MOD51
$NOMOD51
```

Default

```
$MOD51
```

Abbreviation

```
$MO / $NOMO
```

Description

The assembler uses a list of predefined register names. With $NOMOD51 the list will not be used by the assembler.

Example

```
$nomod51
  ; use no predefined list of register names
```

Related Information

Section 2.5, *Registers*

$NOEXTERNALMEMORY

Syntax

`$NOEXTERNALMEMORY`

Abbreviation

`$NOEM`

Description

Certain derivatives like the 8xC751 have no external memory support and therefore do not allow the MOVX instruction. With this control an error message is issued whenever a MOVX instruction is encountered.

$OBJECT / $NOOBJECT

Syntax

$OBJECT(*file***)**
$NOOBJECT

Default

$OBJECT(*sourcefile*.obj**)**

Abbreviation

$OJ / $NOOJ

Description

With the $OBJECT control you can specify an alternative name for the object file. With the $NOOBJECT control no object file will be generated.

Example

```
$oj(myfile.obj)  ; generate object file myfile.obj
```

$OPTIMIZE / $NOOPTIMIZE

Syntax

```
$OPTIMIZE
$NOOPTIMIZE
```

Default

```
$OPTIMIZE
```

Abbreviation

```
$OP / $NOOP
```

Description

With these controls you can turn on or off conditional jump optimization, expansion of generic instructions and jump chain optimizations. This control overrules the **--optimize** (**-O**) command line option.

Example

```
$noop
  ; turn optimization off
  ; source lines
$op
  ; turn optimization back on
  ; source lines
```

Related Information

Assembler option **--optimize**

$PAGELENGTH

Syntax

$PAGELENGTH(*pagelength* **)**

Default

$PAGELENGTH(72)

Abbreviation

$PL

Description

If you generate a list file with the assembler option **--list-file**, the $PAGELENGTH control sets the number of lines per page in the list file.

The argument may be any positive absolute integer expression.

Example

```
$PL(55)        ; page length is 55
```

Related Information

Assembler option **--list-file**

Assembler control **$PAGEWIDTH**

$PAGEWIDTH

Syntax

$PAGEWIDTH(*pagewidth* **)**

Default

$PAGEWIDTH(132)

Abbreviation

$PW

Description

If you generate a list file with the assembler option **--list-file**, the $PAGEWIDTH control sets the width of a page in the list file.

The argument may be any positive absolute integer expression. The default is 132, the minimum is 64 and the maximum is 255. Although greater values for this control are not rejected by the assembler, lines are truncated if they exceed the length of 255.

Example

```
$PW(80)   ; set the pagewidth to 80 characters
```

Related Information

Assembler option **--list-file**

Assembler control **$PAGELENGTH**

$PAGING / $NOPAGING

Syntax

```
$PAGING
$NOPAGING
```

Default

```
$PAGING
```

Abbreviation

```
$PA / $NOPA
```

Description

If you generate a list file with the assembler option **--list-file**, you can use these controls to turn the generation of form feeds in the list file on or off.

Example

```
$nopa
; turn paging off
```

Related Information

Assembler option **--list-file**

Assembler control **$EJECT**

$PRINT / $NOPRINT

Syntax

$PRINT[**(***file***)**]
$NOPRINT

Default

$NOPRINT

Abbreviation

$PR / $NOPR

Description

With the $PRINT control you can generate a list file. Without a file the default filename is *sourcefile*.lst. The $NOPRINT control causes no list file to be generated.

Example

```
$pr(mylist.lst)  ; generate list file mylist.lst
```

Related Information

Assembler option **--list-file**

$REGADDR / $NOREGADDR

Syntax

```
$REGADDR
$NOREGADDR
```

Default

```
$REGADDR
```

Abbreviation

```
$RA / $NORA
```

Description

The `$NOREGADDR` control disallows the use of absolute register addresses as instruction operands. By default absolute registers, like AR0, are allowed as operands.

Example

```
$ra
     mov R1,AR2  ; valid assembly instruction
$nora
     mov R0,AR7  ; AR7 not allowed -> assembler warning W201
```

Related Information

Section 2.5, *Registers*

$REGISTERBANK / $NOREGISTERBANK

Syntax

```
$REGISTERBANK(rb[,rb]...)
$NOREGISTERBANK
```

Default

```
$REGISTERBANK(0)
```

Abbreviation

```
$RB / $NORB
```

Description

With $REGISTERBANK you can specify the register banks used in the current source module. This information is used by the linker to allocate the memory containing the register banks. $NORB specifies that no memory is initially reserved for register banks. The .USING assembler directive also reserves register banks.

Example

```
$rb(0,1,2)    ; reserve register banks 0, 1 and 2
```

Related Information

.USING (Use register bank)

$SAVE / $RESTORE

Syntax

```
$SAVE
$RESTORE
```

Abbreviation

```
$SA / $RE
```

Description

The $SAVE control stores the current value of the $LIST / $NOLIST controls onto a stack. The $RESTORE control restores the most recently saved value; it takes effect starting at the next line. You can nest $SAVE controls to a depth of 16.

Example

```
$nolist
  ; source lines
$save           ; save values of $LIST / $NOLIST

$list

$restore        ; restore value ($nolist)
```

Related Information

Assembler option **--list-file**

Assembler control **$LIST**

$SMALLROM

Syntax

`$SMALLROM`

Abbreviation

`$SR`

Description

When an application fits in a 2K byte block (or no more ROM is supported) LCALL and LJMP instructions can be translated into shorter ACALL and AJMP calls. With this control the conversion will be done automatically. You can also use this control for derivatives (like the 80C751/752) that do not support the LCALL instruction.

A `$TITLE` with no string argument causes the current title to be blank. The title is initially the name of the module. The `$TITLE` control will not be printed in the source listing.

Example

$SMALLROM

; translate LCALL/LJMP to ACALL/AJMP

$TITLE

Syntax

$TITLE([*string*]**)**

Default

No title.

Abbreviation

$TT

Description

The $TITLE initializes the program title to the *string* specified in the operand field. The program title will be printed after the banner at the top of all succeeding pages of the source listing until another $TITLE control is encountered. An exception to this is the first $TITLE control, which sets the title of the first and following pages in the listing until the next $TITLE control is encountered.

A $TITLE with no string argument causes the current title to be blank (this is the default). The $TITLE control will not be printed in the source listing.

Example

```
$TITLE(This is the new title in the list file)
```

Related Information

Assembler option **--list-file**

2.10. Generic Instructions

The assembler supports so-called 'generic instructions'. Generic instructions are pseudo instructions (no instructions from the instruction set). Depending on the situation in which a generic instruction is used, the assembler replaces the generic instruction with appropriate real assembly instruction(s).

The assembler knows the following generic instructions:

CALL/GCALL

- ACALL -> If the target address operand falls within the same 2K page.

- LCALL -> If the target address operand is unknown or outside the same 2K page.

JMP/GJMP

- SJMP -> If the target address operand falls within an 8-bit offset [-128..127].

- AJMP -> If the target address operand falls within the same 2K page.

- LJMP -> If the target address operand is unknown or outside the same 2K page.

GJB

Results in JB if the target address is within the relative range. If the target is not within the relative range, a combination of JNB/LJMP is used.

GJNB

Results in JNB if the target address is within the relative range. If the target is not within the relative range, a combination of JB/LJMP is used.

Chapter 3. Using the C Compiler

This chapter describes the compilation process and explains how to call the C compiler.

The TASKING VX-toolset for 8051 under Eclipse uses the TASKING makefile generator and make utility 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 7.1, *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:
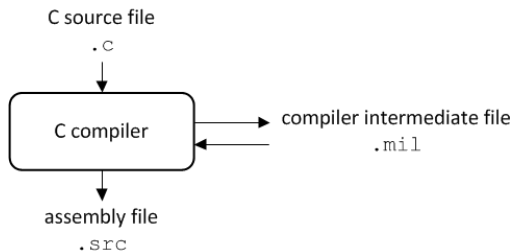
This chapter first describes the compilation process which consists of a *frontend* and a *backend* part. Next it is described how to call the C compiler and how to use its options. An extensive list of all options and their descriptions is included in Section 9.2, *C Compiler Options*. Finally, a few important basic tasks are described, such as including the C startup code and performing various optimizations.

3.1. 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. When there are not enough physical registers, virtual registers are spilled to the stack. Intermediate results of any optimization can live, for some time, on the stack or in physical registers.

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.

3.2. Calling the C Compiler

The TASKING VX-toolset for 8051 under Eclipse uses the TASKING makefile generator and make utility 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 **Behavior** tab of the **C/C++ Build** page of the **Project » Properties for** dialog.

See also Chapter 10, *Influencing the Build Time*.

Select a target processor (core)

Processor options affect the invocation of all tools in the toolset. In Eclipse you need to set them for each configuration. 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 Project wizard (**File » New » TASKING 8051 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 **Configuration** list, select a configuration or select [All configurations].

4. From the **Processor selection** list, select a processor.

To access the 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. From the **Configuration** list, select a configuration or select [All configurations].

4. On the Tool Settings tab, select **C Compiler**.

5. Select the sub-entries and set the options in the various pages.

> Note that the C compiler options are used to create an object file from a 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.

> Note that when you click **Restore Defaults** to restore the default tool options, as a side effect the processor is also reset to its default value on the **Processor** page (**C/C++ Build » Processor**).

You can find a detailed description of all C compiler options in Section 9.2, *C Compiler Options*.

Invocation syntax on the command line:

c51 [[*option*]... [*file*]...]...

3.3. 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 stack pointer and the application C variables.

*   *Exit code.* This controls the close down of the application after the program's main function terminates.

A default startup code is part of the C library. For most situations this should be sufficient, but if the default run-time startup code does not match your configuration, you can add it to your project and modify it.

To add the C startup code to your project

When you create a new project with the New C Project wizard (**File » New » TASKING 8051 C Project**), fill in the dialogs and enable the option **Add startup file(s) to the project** in the following dialog.
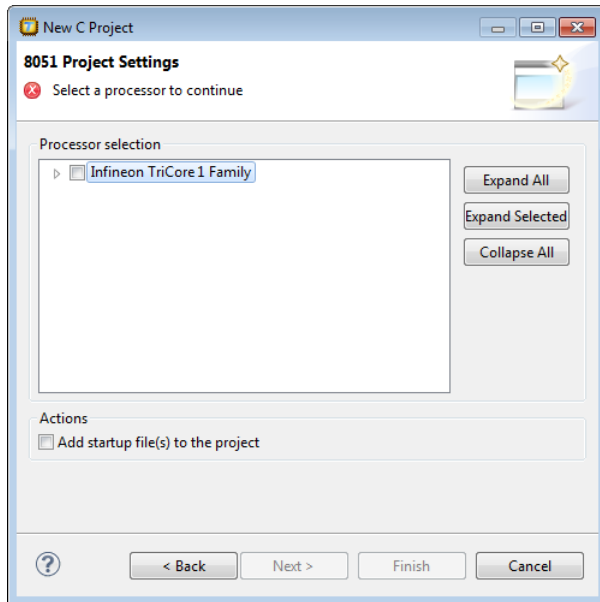
This adds the file `cstart.c` to your project. This file is a copy of `lib/src/cstart.c`. If you do not add the startup code here, the startup code is taken from the C library during link time. You can always add it later with **File » New » Startup Files**.

To change the C startup code in Eclipse manually

1. Double-click on the file `cstart.c`.

    The cstart.c file opens in the editor area.

```
.c cstart.c ⊠
    extern void _init( void );
    #pragma extern  main
    extern int   main( int argc );
    extern void  exit( int );
    #pragma weak    exit
    #pragma extern  _Exit

    extern char __idata _lc_bs[];          /* system stack begin label *
    extern char __xdata _lc_base_pdata[];  /* auxiliary page start addre
    extern char * __data _SP;        /* virtual stack pointer */
    #if __MODEL__ == 'a'
    # define       VSTACK_LABEL    _lc_ue_vstack_pdata
    #endif
    #if __MODEL__ == 'l'
    # define       VSTACK_LABEL    _lc_ue_vstack_xdata
    #endif
    extern char VSTACK_LABEL[];      /* virtual stack end label */

  void __interrupt(0) _start(void)
    {
        P2  = ((unsigned int)_lc_base_pdata) >> 8;     /* needed for whe
        SP  = (unsigned char)_lc_bs;                   /* initialize sta
    #if (__MODEL__ == 'a') || (__MODEL__ == 'l')
        _SP = VSTACK_LABEL-1;                          /* initialize 'vi
    #endif
        _init();                                       /* initialize C v
        exit( main(0) );
    }
```

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.

3.4. 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 Compiler » Include Paths** page in the **C/C++ Build » Settings » Tool Settings** tab of the Project Properties dialog (equivalent to option **--include-directory (-I)**).

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 `C51INC`.

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:

```
c51 -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 `C51INC` 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 `C51INC` and then in the default `include` directory.

3.5. 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 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 3.6, *Compiler Optimizations*.

Invocation syntax on the command line

The invocation syntax on the command line is:

```
c51 -g file.c
```

3.6. 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. From the **Configuration** list, select a configuration or select [All configurations].

4. On the Tool Settings tab, select **C Compiler » Optimization**.

5. 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.

3.6.1. 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).

A CSE can live in a register, on stack or can be recomputed when required.

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.

3.6.2. 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.

Interprocedural register optimization (option -Ob/-OB)

Register allocation is improved by taking note of register usage in functions called by a given function.

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.

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 3.6.3, *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, and name given by the section renaming pragma or option. 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.

Generic assembly optimizations (option -Og/-OG)

A set of target independent optimizations that increase speed and decrease code size.

3.6.3. 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 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.

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

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**

3.7. 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.

3.7.1. 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 15, *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 15, *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 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**.

```
c51 --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.

3.7.2. 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 16, *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 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.

On the command line you can use the option **--misrac**.

```
c51 --misrac={all | number [-number],...]
```

3.8. 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 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:

```
c51 --diag=[format:]{all | number,...]
```

Chapter 4. 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 static profiling.

4.1. 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.

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.

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

Static profiling is described in more detail below in the following section.

4.2. Profiling at Compile Time (Static Profiling)

Static profiling can be used to determine which parts of a program take most of the execution time.

Overview of steps to perform

To obtain a profile, 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, you can use the Blink example as described below.

1. From the **File** menu, select **Import...**

   The Import dialog appears.

2. Select **TASKING C/C++ » TASKING 8051 Example Projects** and click **Next**.

3. In the **Example projects** box, disable all projects except `xc800-blink` and `xc800-blink-start`.

4. Click **Finish**.

   The projects should now be visible in the C/C++ Projects view.

4.2.1. 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 xc800-blink-start 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 (🐱).

4.2.2. 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 `xc800-blink-start.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.

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 ($\bigtriangledown$) 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 or cstart. 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
{ⁱⁱⁱ}	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.

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 Projects box, select the project for which you want to see profiling information.

3. In the **Profiling Type** group box, select **Static Profiling**.

4. In the **Static Profiling File** group box, enable the option **Use default**.

   By default, the file *project*.xprof is used (xc800-blink-start.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).

5. Click **OK** to finish.

Chapter 5. 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:
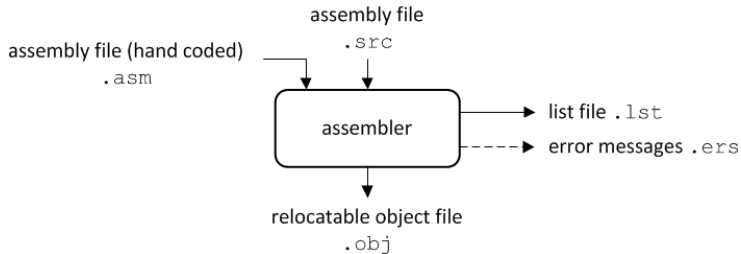
The following information is described:

* 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 9.3, *Assembler Options*.

* The various assembler optimizations.

* How to generate a list file.

* Types of assembler messages.

5.1. Assembly Process

The assembler generates relocatable output files with the extension `.obj`. 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

* 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 by means of a built-in macro preprocessor. See Section 2.8, *Macro Preprocessing* for more information.

5.2. Calling the Assembler

The TASKING VX-toolset for 8051 under Eclipse uses the TASKING makefile generator and make utility 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) (![icon]). 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 (![icon]).

    To build individual projects incrementally, select **Project » Build** *project*.

- Rebuild Project (![icon]). 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 **Behavior** 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 need to set them for each 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 page appears.

3. From the **Configuration** list, select a configuration or select [All configurations].

4.   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.   From the **Configuration** list, select a configuration or select [All configurations].

4.   On the Tool Settings tab, select **Assembler**.

5.   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.

> Note that when you click **Restore Defaults** to restore the default tool options, as a side effect the processor is also reset to its default value on the **Processor** page (**C/C++ Build » Processor**).

You can find a detailed description of all assembler options in Section 9.3, *Assembler Options*.

Invocation syntax on the command line:

as51 [[*option*]... [*file*]...]...

The input file must be an assembly source file (.asm or .src).

5.3. How the Assembler Searches Include Files

When you use include files (with the %INCLUDE macro preprocessing function), you can specify their location in several ways. The assembler searches the specified locations in the following order:

1. If the %INCLUDE function 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 option **--include-directory (-I))**.

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 AS51INC.

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:

```
as51 -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 `AS51INC` and then in the default `include` directory.

5.4. 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. From the **Configuration** list, select a configuration or select [All configurations].

4. On the Tool Settings tab, select **Assembler » Optimization**.

5. 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.

By default this option is enabled. If you turn off this optimization, generic instructions are not allowed. In that case you have to use hardware instructions.

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.

5.5. 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

The following command generates the list file `test.lst`:

```
as51 -l test.asm
```

See Section 12.1, *Assembler List File Format*, for an explanation of the format of the list file.

5.6. 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:

```
as51 --diag=[format:]{all | number,...]
```

Chapter 6. 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 (`.obj` 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:
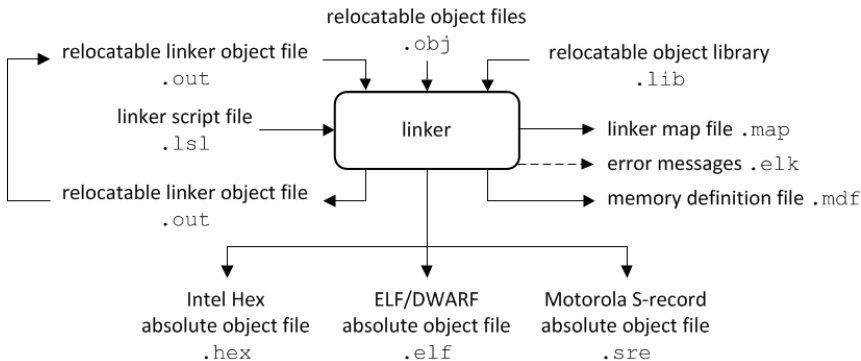
This chapter first describes the linking process. Then it describes how to call the linker and how to use its options. An extensive list of all options and their descriptions is included in Section 9.4, *Linker Options*.

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.

6.1. Linking Process

The linker combines and transforms relocatable object files (`.obj`) 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 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
	• 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).
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.
Relocatable object file	Object code in which addresses are represented by symbols and thus relocatable.
Relocation	The process of assigning absolute addresses.

Term	Definition
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.

6.1.1. 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 (`.obj`) or libraries (`.lib`) 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.

6.1.2. 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 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 6.7, *Controlling the Linker with a Script*.

6.2. 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 (![icon]).

To build individual projects incrementally, select **Project » Build** *project*.

• Rebuild Project (![icon]). 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 **Behavior** 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. From the **Configuration** list, select a configuration or select [All configurations].

4. On the Tool Settings tab, select **Linker**.

5. Select the sub-entries and set the options in the various pages.

> Note that when you click **Restore Defaults** to restore the default tool options, as a side effect the processor is also reset to its default value on the **Processor** page (**C/C++ Build » Processor**).

You can find a detailed description of all linker options in Section 9.4, *Linker Options*.

Invocation syntax on the command line:

lk51 [[*option*]... [*file*]...]...

When you are linking multiple files, either relocatable object files (.obj) or libraries (.lib), it is important to specify the files in the right order. This is explained in Section 6.3, *Linking with Libraries*.

Example:

```
lk51 -d51.lsl test.obj
```

This links and locates the file test.obj and generates the file test.elf.

6.3. Linking with Libraries

There are two kinds of libraries: system libraries and user libraries.

System library

System libraries are stored in the directories:

```
<8051 installation path>\lib\c51     (libraries for core c51)
<8051 installation path>\lib\scr2g   (libraries for core scr2g)
```

An overview of the system libraries is given in the following table:

Libraries	Description
c51*m*{r | s}[*b*].lib	C libraries for each model *m*: s (small), a (aux), l (large) Optional letters: r = reentrant s = static *b* = bank number 0, 1, 2, 3
fp51*m*{r | s}[t].lib	Floating-point libraries for each model *m*: s (small), a (aux), l (large) Optional letters: r = reentrant s = static t = trapping (control program option **--fp-model=+trap**)
rt51.lib	Run-time library

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 c51ss0.lib, type:

```
lk51 --library=c51ss0 -d51.lsl test.obj
```

User library

You can create your own libraries. Section 7.4, *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:

```
lk51 –d51.lsl start.obj mylib.lib
```

If the library resides in a sub-directory, specify that directory with the library name:

```
lk51 –d51.lsl start.obj mylibs\mylib.lib
```

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:

```
lk51 --first-library-first a.lib test.obj b.lib
```

If the file `test.obj` calls a function which is both present in `a.lib` and `b.lib`, normally the function in `b.lib` would be extracted. With this option the linker first tries to extract the symbol from the first library `a.lib`.

Note that routines in `b.lib` that call other routines that are present in both `a.lib` and `b.lib` are now also resolved from `a.lib`.

6.3.1. 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 option **--library-directory (-L)**). 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 variable `LIBC51`.

3. When the linker did not find the library, it tries the default `lib\c51` directory or `lib\scr2g` directory relative to the installation directory, depending on the core architecture.

User library

If you use your own library, the linker searches the library in the current directory only.

6.3.2. How the Linker Extracts Objects from Libraries

A library built with the TASKING archiver **ar51** 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:

```
lk51 mylib.lib
```

nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through `mylib.lib`.

It is possible to force a symbol as external (unresolved symbol) with the option **--extern** (**-e**):

```
lk51 --extern=main mylib.lib
```

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.lib`. This process repeats until no new unresolved symbols are found.

6.4. Incremental Linking

With the TASKING linker it is possible to link incrementally. Incremental linking means that you link some, but not all `.obj` 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 `.obj` files as the `.out` file you had created with the first invocation.

> Incremental linking is only possible on the command line.

```
lk51 -d51.lsl --incremental test1.obj -otest.out
lk51 -d51.lsl test2.obj test.out
```

This links the file `test1.obj` and generates the file `test.out`. This file is used again and linked together with `test2.obj` 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 `.obj` 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.

6.5. 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.

Add a data object in Eclipse

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).

On the command line you can add raw data to your application with the linker option **--import-object**.

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 2-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>
__rom extern char   _lc_ub_my_mp3; /* linker labels */
__rom extern char   _lc_ue_my_mp3;
__rom 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 type qualifier `__rom`, otherwise the linker cannot bind your linker symbols.

If you want to use the export functionality of Eclipse, the binary file has to be part of your project.

6.6. 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.

Note that this optimization only affects unrestricted sections that require an initialization action in the copy table. The affected sections get a clustered restriction. Unrestricted sections are sections that do not have their absolute location or their relative location to other sections restricted. See also Define the mutual order of sections in an LSL group in Section 14.8.2, *Creating and Locating Groups of Sections*.

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.

6.7. 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.

6.7.1. 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 14, *Linker Script Language (LSL)*.

6.7.2. 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`.

Because an 8051 project is part of a TriCore project you only need to specify an LSL file to the TriCore project.

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.
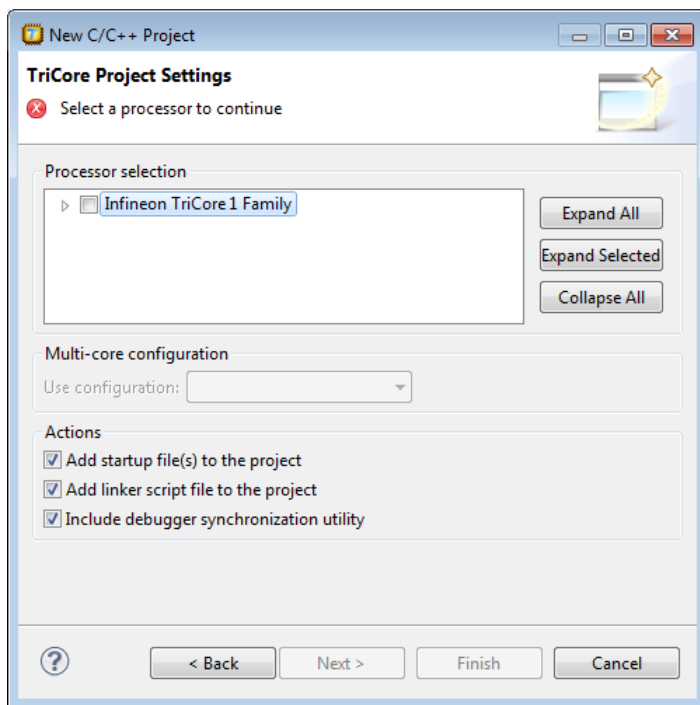
3. Enable the option **Add linker script file to the project** and click **Finish**.

    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)**.
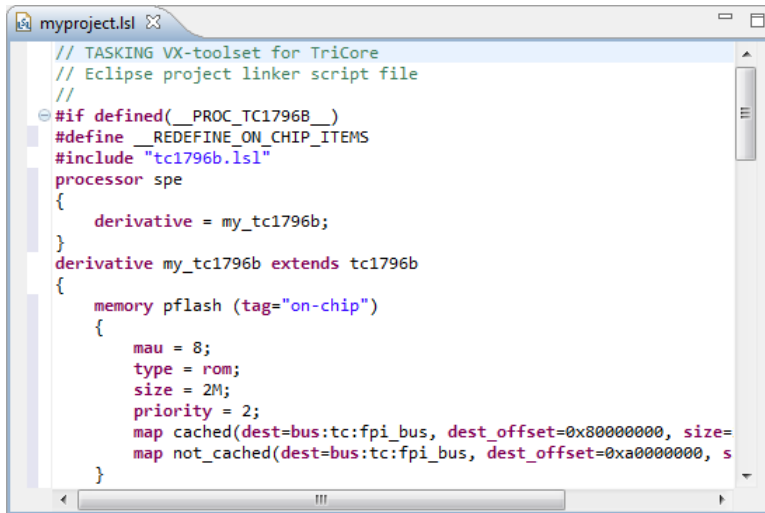
To change the Linker Script File in Eclipse

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 ![save icon] 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**).

6.7.3. 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 stack and the heap.

This specification is normally written by Altium. Altium supplies LSL files in the `include.lsl` directory. The file `arch51.lsl` defines the architecture for generic 8051, `arch_scr2g.lsl` defines the architecture for AURIX 2G SCR, it extends the generic 8051 architecture.

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.

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

```
architecture c51
{
    // Specification of the c51 core architecture.
    // Written by Altium.
}

derivative X          // derivative name is arbitrary
{
    // Specification of the derivative.
    // Written by Altium.
    core xc800        // always specify the core
    {
        architecture = c51;
    }

    bus idata_bus
    {
        // internal bus
    }

    // internal memory
}

processor mpe         // 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 mpe:xc800:xdata    // section layout
{
    // section placement statements
```

```
    // sections are located in address space 'xdata'
    // of core 'xc800' of processor 'mpe'
}
```

Overview of LSL files delivered by Altium

Altium supplies the following LSL files in the directory `include.lsl`.

LSL file	Description
`arch51.lsl`	Defines the c51 architecture and contains a default section layout.
`arch_scr2g.lsl`	Defines the scr2g architecture, it extends the c51 architecture. It includes the file `arch51.lsl`.
`derivative.lsl`	Derivative LSL file. It contains memory definitions and section_setup definitions. It includes one of the files `arch51.lsl` or `arch_scr2g.lsl`.
`default.lsl`	It includes a `derivative.lsl` file based on the target processor you specified with control program option **--cpu**. You can add this file to your project and adapt it to your needs, or create your own LSL file.

The linker uses the file `default.lsl`, unless you specify another file with the linker option **--lsl-file** (**-d**).

6.7.4. 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 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, separate spaces for code and data. 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 `c51` as defined in `arch51.lsl`.

Space	Id	MAU	Description
data	1	8	Direct addressable data (on-chip), contains definitions for a heap.
idata	2	8	Indirect addressable space (on-chip), contains definitions for the stack.
bdata	3	8	Bit-addressable data (on-chip RAM).
bit	4	1	Bit address space.
xdata	5	8	External data, contains definitions for a virtual stack and a heap.
pdata	6	8	Paged data, mapped in one 256 bytes page. Contains definitions for a virtual stack and a heap.
code	7	8	Code address space, specifies the start address at the beginning of the vector table.

The c51 architecture in LSL notation

The best way to program the architecture definition, is to start with a drawing. The figure below shows a part of the architecture `c51`
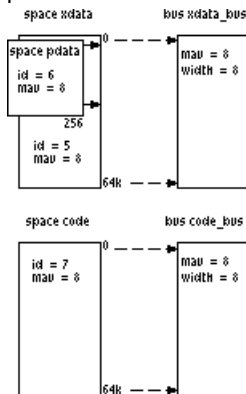
The figure shows three address spaces called `xdata`, `pdata` and `code`. The address space `pdata` is a subset of the address space `xdata`. All address spaces have attributes like a number that identifies the logical space (id), a MAU and an alignment. In LSL notation the definition of these address spaces look as follows:

```
space xdata
{
    id  = 5;
    mau = 8;
    map (size=64k, dest=bus:xdata_bus);
}

space pdata
{
```

```
    id  = 6;
    mau = 8;
    map (size=256, dest_offset=__BASE_PDATA & 0xff00, dest=space:xdata);
}

space code
{
    id  = 7;
    mau = 8;
    map (size=64k, dest=bus:code_bus);
}
```

The keyword `map` corresponds with the dotted lines 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 internal buses named `xdata_bus` and `code_bus` must be defined in LSL:

```
bus xdata_bus
{
    mau = 8;
    width = 8;  // there are 8 data lines on the bus
}

bus 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 c51
{
    // All code above goes here.
}
```

6.7.5. 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 xc800
{
    architecture = c51;
}
```

Bus

Each derivative can contain a bus definition for connecting external memory. In this example, the bus xdata_bus maps to the bus xdata_bus defined in the architecture definition of core xc800:

```
bus xdata_bus
{
    mau = 8;
    width = 8;
    map (dest=bus:xc800:xdata_bus, dest_offset=0, size=256);
}
```

Memory

Memory is usually described in a separate memory definition, but you can specify on-chip memory for a derivative. For example:

```
memory internal_code_rom
{
    type = rom;
    size = 2k;
    mau = 8;
    map(dest = bus:code_bus, size=2k,
        dest_offset = 0x00);  // src_offset is zero by default)
}
```

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
}
```

6.7.6. 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.

6.7.7. The Memory Definition

Once the core architecture is defined in LSL, you may want to extend the processor with 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
}
```
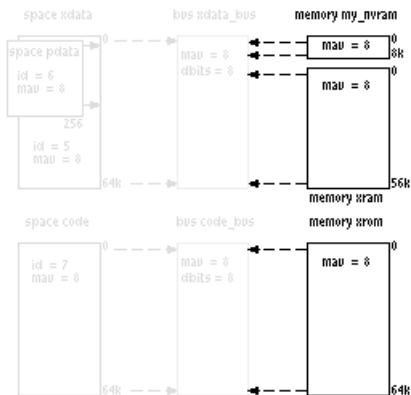

Suppose your embedded system has 56kB of external RAM, named `xram`, 8 kB of external NVRAM, named `my_nvram` and 64kB of external ROM, named `xrom` (see figure above.) `xram` and `my_nvram` are connected to the bus `xdata_bus` and `xrom` is connected to the bus `code_bus`. In LSL this looks like follows:

```
memory my_nvram
{
    mau = 8;
    type = nvram;
    size = 8k;
```

```
    map (dest = bus:xc800:xdata_bus, src_offset = 0, dest_offset = 0,
        size=8k);
}

memory xram
{
    mau = 8;
    type = ram;
    size = 56k;
    map (dest = bus:xc800:xdata_bus, src_offset = 0, dest_offset = 8k,
        size=56k);
}

memory xrom
{
    mau = 8;
    type = rom;
    size = 64k;
    map (dest = bus:xc800:code_bus, src_offset = 0, dest_offset = 0,
        size=64k);
}
```

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_nvram`), 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.

6.7.8. 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 data=non_volatile
#pragma noclear
int  battery_backup_tag;
int  battery_backup_invok;
#pragma clear
#pragma endsection

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` and `#pragma endsection` 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 "`data_`*variable*" of section type "`data`" carrying section attribute "`clear`" to store uninitialized data objects. The section type and attribute tell the linker to locate the section in address space data and that the section content should be filled with zeros at startup.

As a result of the `#pragma section data=non_volatile`, the data objects between the pragma pair are placed in a section with the name "`non_volatile`". Note that the compiler sets the "`clear`" attribute. 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_nvram` from the example in Section 6.7.7, *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 `near`:

```
section_layout ::data
{
    // Section placement statements
}
```

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 `non_volatile`. All other sections are located using the defaults specified in the architecture definition. Section `non_volatile` should be placed in non-volatile ram. To achieve this, the run address refers to our non-volatile memory called `my_nvram`.

```
group ( run_addr = mem:my_nvram )
{
    select "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 14, *Linker Script Language (LSL)*.

6.8. 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 begin of the stack or heap or copy table.

`__lc_ue_name`
 	
`__lc_e_name`
	End of section *name*. Also used to mark the end 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.

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.

At C level, all linker labels start with one leading underscore (the compiler adds an extra underscore).

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.

Additionally, the linker script file defines the following symbols:

Symbol	Description
`__lc_bs`	Begin of stack. Same as `__lc_ub_stack`.
`__lc_base_pdata`	Start of paged data. Alias for `__BASE_PDATA & 0xff00`
`__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.

Example: refer to a label with section name with dots from C

Suppose a section has the name `.text`. When you want to refer to the begin of this section you have to replace all dots in the section name by underscores:

```
#include <stdio.h>
extern void * _lc_ub__text;
```

```
int main(void)
{
    printf("The function main is located at %x\n",
           &_lc_ub__text);
}
```

Example: refer to the stack

Suppose in an LSL file a stack section is defined with the name "`stack`" (with the keyword `stack`). You can refer to the begin and end of the stack from your C source as follows (labels have one leading underscore):

```
#include <stdio.h>
extern char _lc_ub_stack[];
extern char _lc_ue_stack[];
int main(void)
{
  printf( "Size of stack is %d\n",
          _lc_ue_stack - _lc_ub_stack );
}
```

From assembly you can refer to the end of the stack with:

```
  .extrn idata(__lc_ue_stack)   ; end of stack
```

6.9. 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

The following command generates the map file `test.map`:

```
lk51 --map-file test.obj
```

With this command the map file `test.map` is created.

See Section 12.2, *Linker Map File Format*, for an explanation of the format of the map file.

6.10. 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:

```
lk51 --diag=[format:]{all | number,...]
```

Chapter 7. Using the Utilities

The TASKING VX-toolset for 8051 comes with a number of utilities:

cc51   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.

amk   A make utility 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. It supports parallelism which utilizes the multiple cores found on modern host hardware.

mk51   A make utility for backwards compatibility with older versions of the toolset. Not recommended for new projects.

ar51   An archiver. With this utility you create and maintain library files with relocatable object modules (`.obj`) generated by the assembler.

hldump51   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.

expire51   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.

7.1. 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 sources without the need to invoke the compiler, assembler and linker manually.

Eclipse uses the control program to call the C compiler, assembler and linker, but you can call the control program from the command line. The invocation syntax is:

cc51 [[*option*]... [*file*]...]...

Recognized input files

- 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 `.lib` suffix are interpreted as library files and are passed to the linker.

- Files with a `.obj` 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-*** (**-Wc**, **-Wa**, **-Wl**) to pass arguments directly to tools.

For a complete list and description of all control program options, see Section 9.5, *Control Program Options*.

Example with verbose output

```
cc51 --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\c51" -Ms --registerbank=0 -o cc3248a.src test.c
+ "path\as51" -o cc3248b.obj cc3248a.src
+ "path\lk51" -o test.elf -D__CPU__=51 --map-file
    cc3248b.obj -lc51ss0 -lfp51ss -lrt51"
```

The control program produces unique filenames for intermediate steps in the compilation process (such as `cc3248a.src` and `cc3248b.obj` 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

```
cc51 --pass-c=-Oc test.c
```

The option **-Oc** is directly passed to the compiler.

7.2. Make Utility amk

amk is a make utility that you can use to maintain, update, and reconstruct groups of programs. **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".

7.2.1. 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.obj : test.c
```

This states that target test.obj 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.obj :
  cc51 test.c   # leading white space
```

Command rules may contain dependencies too. Combining the above for example yields:

```
test.obj : test.c
  cc51 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.obj` matches the pattern '`%.obj`', 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.obj filter.obj

all: $(objects)

$(objects): %.obj: %.c
    cc51 -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.obj: test.c
    cc51 -c test.c -o test.obj
    echo the stem is test

filter.obj: filter.c
    cc51 -c filter.c -o filter.obj
    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
.DONE	When the make utility has finished building the specified targets, it continues with the rules following this target.

Target	Description
`.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`.

7.2.2. 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.

7.2.3. 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. 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:.obj"=.d")
```

---

[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.

7.2.4. 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.obj 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.obj readme.txt .project output.c}
```

results in:

```
test.obj 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}
```

results in:

```
test.c test.h filter.c filter.h output.c output.h
```

7.2.5. 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. 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
```

7.2.6. 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).

7.2.7. 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.
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.

7.2.8. 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 9.7, *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.

7.3. Make Utility mk51

This make utility is for backwards compatibility with older versions of the toolset. It is not recommended for new projects. Use **amk** instead.

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 **mk51** 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 `mk51.mk` which contains predefined rules and macros. See Section 7.3.2, *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 `.obj` files and libraries that must be linked together. The `.obj` 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
          c51 test.c              # rule

test.obj : test.src
          as51 test.src

test.elf : test.obj
          lk51 test.obj –o test.elf --map-file –lc51ss0 –lfp51ss –lrt51
```

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 **mk51** with one of the following lines:

```
mk51 test.elf
```

```
mk51 -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, **mk51** 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.obj`.

2. If `test.obj` does not exist or is out-of-date, the make utility first tries to build this file and reads from the makefile that `test.obj` depends on `test.src`.

3. If `test.src` does exist, the make utility now creates `test.obj` by executing the rule for it: `as51 test.src`.

4. There are no other files necessary to create `test.elf` so the make utility now can use `test.obj` to create `test.elf` by executing the rule: `lk51 test.obj -o test.elf ...`

The make utility has now built `test.elf` but it only used the assembler to update `test.obj` and the linker to create `test.elf`.

If you compare this to the control program:

```
cc51 test.c
```

This invocation has the same effect but now *all files* are recompiled (assembled, linked and located).

7.3.1. Calling the Make Utility

You can only call the make utility from the command line. The invocation syntax is:

mk51 [[*option*]... [*target*]... [*macro=def*]...]

For example:

```
mk51 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 **mk51** invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is inherited by subordinate **mk51**'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 9.6, *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.

7.3.2. Writing a Makefile

In addition to the standard makefile `makefile`, the make utility always reads the makefile `mk51.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 `mk51.mk` file) you can prevent the make utility from reading `mk51.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
```

7.3.2.1. 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.obj demo.obj final.obj
```

You can now can specify the target you want to build to the make utility. The following three invocations all have the same effect:

```
mk51
mk51 all
mk51 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.obj : 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.
.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.

Target	Description
`.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 `mk51.mk`. If you specify this target with dependencies, these are added to the existing `.SUFFIXES` target in `mk51.mk`. If you specify this target without dependencies, the existing list is cleared.

7.3.2.2. 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
         c51 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 **mk51** to execute multiple command lines in one shell environment. This is accomplished with the token combination '**;**'. For example:

```
cd c:\Tasking\bin ;\
mk51 -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:

```
lk51 -o $@ -f <<EOF
     $(separate "\n" $(match .obj $!))
     $(separate "\n" $(match .lib $!))
     $(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: `lk51 -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.obj   :
             cc51 -c $<
```

Read this as: to build a file with extension `.obj` 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 `mk51.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  =     -lc51ss0 -lfp51ss -lrt51    # macro

prog.elf:  prog.obj sub.obj
     lk51  prog.obj sub.obj $(LIB) -o prog.elf

prog.obj:  prog.c inc.h
     c51   prog.c
     as51  prog.src

sub.obj:   sub.c inc.h
     c51   sub.c
     as51  sub.src
```

This makefile says that `prog.elf` depends on two files `prog.obj` and `sub.obj`, 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 `mk51.mk`) to perform the same job.

```
LDFLAGS = -lc51ss0 -lfp51ss -lrt51      # macro used by implicit rules
prog.elf: prog.obj sub.obj              # implicit rule used
prog.obj: prog.c inc.h                  # implicit rule used
sub.obj:  sub.c inc.h                   # implicit rule used
```

7.3.2.3. 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 **mk51**. 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 **mk51** (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 **mk51**'s, but it is also available to these invocations as an environment variable.

Macro	Description

PRODDIR	Holds the name of the directory where **mk51** 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 **mk51** 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.

7.3.2.4. 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 .obj prog.obj sub.obj mylib.lib)
```

yields:

```
prog.obj sub.obj
```

$(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.obj sub.obj)
```

results in:

```
prog.obj
sub.obj
```

Function arguments may be macros or functions themselves. So,

```
$(separate "\n" $(match .obj $!))
```

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 cc51 test.c)
```

When the file `test.c` exists, it yields:

```
cc51 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 cc51 test.c)
```

$(addprefix *prefix*, *argument* ...)

The `addprefix` function adds a prefix to its arguments. It is used in `mk51.mk` for invocation of the control program to pass arguments directly to a tool.

Example:

```
cc51 $(addprefix -Wc, -g1 -O2) test.c
```

yields:

```
cc51 -Wc-g1 -Wc-O2 test.c
```

7.3.2.5. 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
```

7.3.2.6. 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
          cc51 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.

7.4. Archiver

The archiver **ar51** is a program to build and maintain your own library files. A library file is a file with extension `.lib` and contains one or more object files (`.obj`) 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:
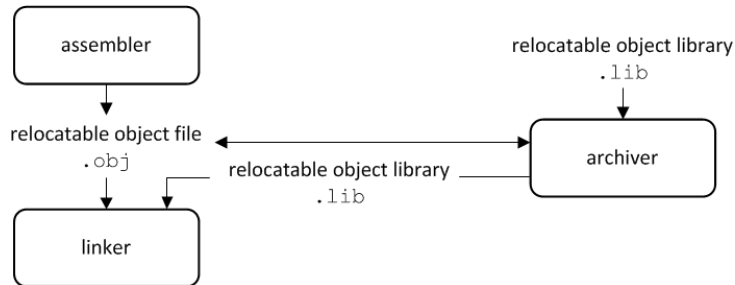
The linker optionally includes object modules from a library if that module resolves an external symbol definition in one of the modules that are read before.

7.4.1. 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 an 8051 absolute ELF file, you can choose to create a library. You do this when you create a new project with the New C Project wizard.

1. From the **File** menu, select **New » TASKING 8051 C Project**.

   The New C Project wizard appears.

2. Enter a project name.

3. In the **Project type** box, select **TASKING 8051 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:

```
ar51 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 -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}**	

Description	Option	Sub-option
Replace only newer modules	**-u**	
Verbose	**-v**	
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*	

For a complete list and description of all archiver options, see Section 9.8, *Archiver Options*.

7.4.2. 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.lib` and add the object modules `cstart.obj` and `calc.obj` to it:

```
ar51 -r mylib.lib cstart.obj calc.obj
```

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.)

```
ar51 -r mylib.lib mod3.obj
```

Print a list of object modules in the library

To inspect the contents of the library:

```
ar51 -t mylib.lib
```

The library has the following contents:

```
cstart.obj
calc.obj
mod3.obj
```

Move an object module to another position

To move `mod3.obj` to the beginning of the library, position it just before `cstart.obj`:

```
ar51 -mb cstart.obj mylib.lib mod3.obj
```

Delete an object module from the library

To delete the object module `cstart.obj` from the library `mylib.lib`:

`ar51 -d mylib.lib cstart.obj`

Extract all modules from the library

Extract all modules from the library `mylib.lib`:

`ar51 -x mylib.lib`

7.5. HLL Object Dumper

The high level language (HLL) dumper **hldump51** 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.

7.5.1. Invocation

Command line invocation

You can call the HLL dumper from the command line. The invocation syntax is:

hldump51 [*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 9.9, *HLL Object Dumper Options*. With `hldump51 --help` you will see the options on `stdout`.

7.5.2. 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. Section dump (disassembly)

4. HLL symbol table

5. Assembly level symbol table

6. Debug control flow section

With the option **--dump-format=**_flag_ you can control which parts are shown. By default, all parts are shown, except for part 6.

Example

Suppose we have a simple "Hello World" program in a file called `hello.c`. We call the control program as follows:

```
cc51 -g -t --control-flow-info hello.c
```

Option **-g** tells to include DWARF debug information. Option **-t** tells to keep the intermediate files. Option **--control-flow-info** adds control flow information to the output file. This command results (among other files) in the file `hello.elf` (the absolute object file).

We can dump information about the object file with the following command:

```
hldump51 -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
000004ef       9       1 text       code_main
00000037       6       1 bss        data_world
0000002c      11       1 bss        data__1_str

---------- Section dump ----------

                                   .rseg '$interrupt_0' code, at(0x00000000)
00000000 02 04 4f   _$interrupt_0: LJMP     __start
                                   ; End of section

                                   .rseg 'code_main' code, at(0x000004ef)
000004ef 74 37       _main:        MOV      A, #55
000004f1 f5 49                     MOV      _$varargs_printf, A
000004f3 7f 2c                     MOV      R7, #44
000004f5 02 04 f8                  LJMP     _printf
                                   ; End of section

              .rseg '[data__1_str]' data, at (0x0537)
              .db 48,65,6c,6c,6f,20,25,73,21,0a,00                ; Hello %s!..
              ; End of section

              .rseg '[data_world]' data, at (0x0542)
              .db 77,6f,72,6c,64,00                               ; world.
              ; End of section

---------- HLL symbol table ----------
```

```
Address         Size HLL Type              Name
3:00000008     36 struct _iobuf __data _iob[4] [_iob.c]
3:00000037      6 char         __data world[6] [hello.c]
3:0000003d      9 struct       __data _dbg_request [dbg.c]
10:0000044f    22 void          __static __data __bank0__interrupt _start()
10:000004ef     9 void          __static __data __bank0 main()


---------- assembly level symbol table ----------

Address      Size     Type Name
0:00000000
0:00000000              _Exit
0:00000000              _dbg_request
0:00000000              main
0:00000000              hello.src
1:00000000              data__dbg_request
1:00000000              data__iob
1:00000000              data_world
1:00000037         data _world
7:000004ef         code _main
7:000004f8         code _printf


---------- Debug control flow section ----------
start offset : 0
start address: 0x000004ef
code size    : 9
#entries     : 0
```

Module list

This part lists all modules (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.

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 .db, .dw, .dl kind of directives, depending on the size of the data. RAM sections are represented with .ds 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 57 ([data__1_str]):
00000537 48 65 6c 6c 6f 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. The number in front of the address indicates the memory type qualifier. For example, `3:` indicates `__data`.
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:

```
3:00000037      6 char          __data world[6] [hello.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:

```
3:00000037      6 char          __data world[6] [hello.c]
3:00000037      1   char
3:00000038      1   char
3:00000039      1   char
3:0000003a      1   char
3:0000003b      1   char
3:0000003c      1   char
```

HLL struct and union symbols are listed by default without fields. For example:

```
3:0000003d      9 struct         __data _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:

```
3:0000003d      9 struct         __data _dbg_request [dbg.c]
3:0000003d      2   int                 _errno
3:0000003f      1   enum                nr
3:00000040      6   union               u
3:00000040      2     struct              exit
3:00000040      2       int                 status
3:00000040      3     struct              open
3:00000040      1       const char  __data * pathname
3:00000041      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:

```
10:0000047e    66 int  __static __data __bank0 fclose(struct _iobuf __data * fp)
```

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:

```
3:00000100     4 int               count [file.c, somefunc()]
3:00000104     4 int               count [x\a.c]
3:00000108     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. The number in front of the address indicates the memory space as specified in the LSL file. For example, 7: indicates space `code`.
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.

7.6. Expire Cache Utility

With the utility **expire51** 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 10.4, *Compiler Cache*.

The invocation syntax is:

expire51 [*option*]... *cache-directory*

The compiler cache is present in the directory c51cache under the specified *cache-directory*.

For a complete list and description of all options, see Section 9.10, *Expire Cache Utility Options*. With expire51 --help you will see the options on stdout.

Examples

To remove all files older than seven days, enter:

expire51 --days=7 "*installation-dir*\mproject\.cache"

To reduce the compiler cache size to 4 MB, enter:

expire51 --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:

expire51 --megabytes=0 "*installation-dir*\mproject\.cache"

Chapter 8. Using the Debugger

This chapter describes the debugger and how you can run and debug a C application. This chapter only describes the TASKING specific parts.

8.1. 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.

8.2. Debugging an 8051 Project

In order to debug an 8051 project, follow the steps below.

1.   Create an 8051 project (for example, `myproject`), as explained in the *Getting Started with the TASKING VX-toolset for TriCore*. Enable at least the **Debug** configuration.

2.   (Optional) Build the 8051 project. This step is optional because the `.out` file is built automatically when the project is referenced and built from a TriCore project. See step 5.

     This results in a linked output file (`.out`).

3.   Create a TriCore project, as explained in the *Getting Started with the TASKING VX-toolset for TriCore*.

4.   In the TriCore project, make a project reference to the 8051 project.

5.   Build the TriCore project.

     This builds the referenced 8051 project and creates the `.out` file in the Debug directory of the 8051 project. Furthermore this creates a TriCore ELF file and an 8051 ELF file in the Debug directory of the TriCore project.

6.   Make the 8051 project the active project.

7. Create a debug configuration for the 8051 project, as explained in Section 8.3, *Creating a Customized Debug Configuration*.

8. Start the debugger. From the **Debug** menu select **Debug** *project*. Alternatively you can click the 🔆 button in the main toolbar.

Example

An SCR XC800 LED example (`xc800-blink`) is delivered with the product for the TriBoard TC26x and is referenced by the TriCore project `xc800-blink-start`. Use the Import wizard (**File » Import » TASKING C/C++ » TASKING 8051 Example Projects**) to import the projects into the workspace.

8.3. 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**.

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 **TASKING 8051 Simulator**.

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. The information in this tab is based on the Debug Target Configuration (DTC) files.
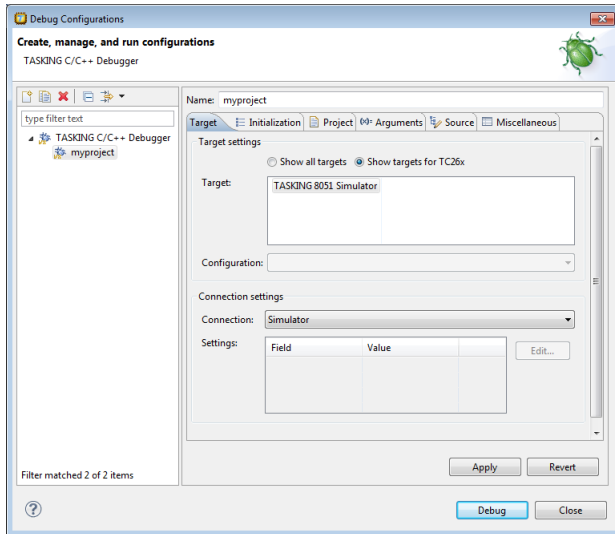
Initialization tab

On the **Initialization** tab enable one or more of the following options:
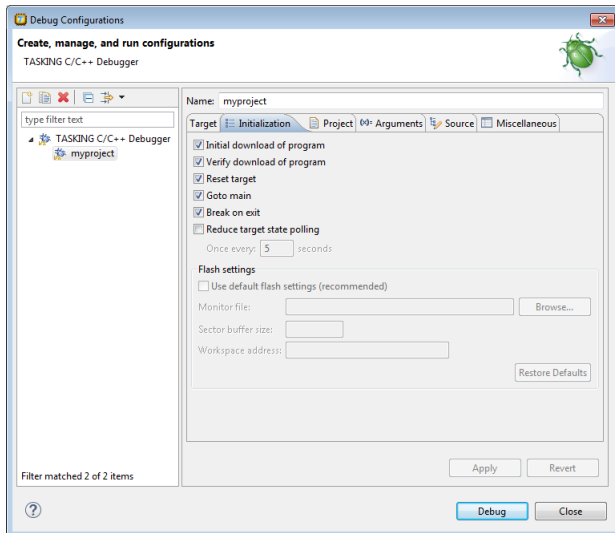
* **Initial download of program**

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.

- **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.

Initialization tab: Flash settings

- **Use default flash settings (recommended)**

  By default, the flash settings are derived from the .dtc file for the chosen target processor. So, when you change processors the flash settings change automatically. If you do not want that, you can specify your own flash settings. You can click **Restore Defaults** to restore the default flash settings.

- **Monitor file**

  Filename of the monitor, usually an Intel Hex or S-Record file.

- **Sector buffer size**

  Specifies the buffer size for buffering a flash sector.

- **Workspace address**

  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(s) which are used when you choose this configuration.
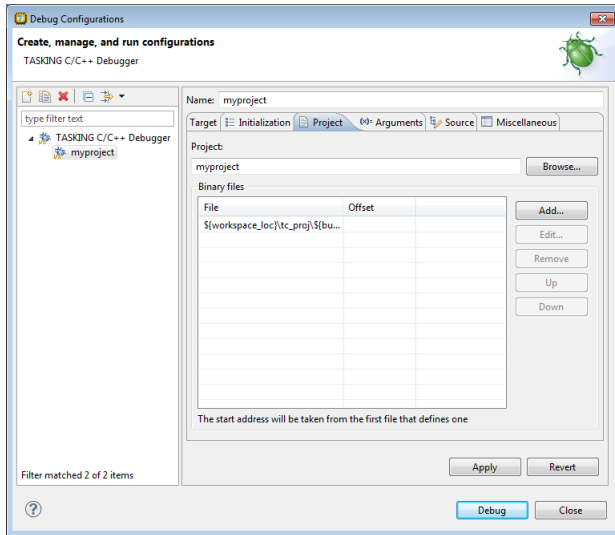
- In the **Project** field, you can choose the project for which you want to make a debug configuration. Because the project myproject is the active project, this project is filled in automatically. Click the Browse... button to select a different project. Only the *opened* projects in your workbench are listed.

- In the **Binary files** group box, you can choose one or more binary files to debug. The file myproject.elf is automatically selected from the active project.

    The order of the binary files matters. Use the **Up** and **Down** buttons to change the order. If there are multiple files, the application start address is taken from the first file that defines one. An ELF file always defines one, whereas Hex files may not. Try to avoid address overlaps.

    Note that conflicts between symbols could arise, for example when you download two ELF files that both contain the function main(). When you download multiple files, we recommend that the first binary file is an ELF file that contains the startup code and main() and that the other files are auxiliary Hex files.

    To add a binary file

    1. Click **Add...** to add a binary file.

        The Add Binary File dialog appears.
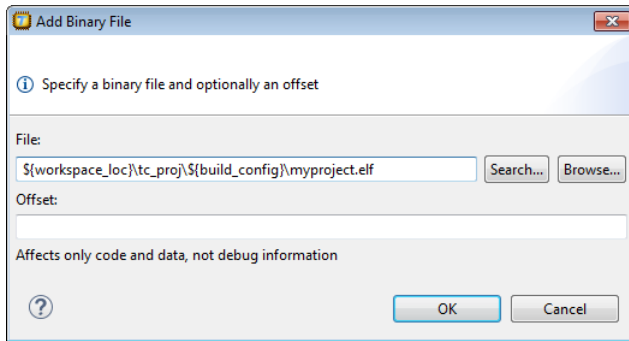
2. Specify the binary file, use the **Search...** button to select one from the active project, or use the **Browse...** button to search the file system.

3. Optionally, specify an address offset. The value will be added to all target addresses in the binary file.

   Note that the address offset will be applied only to code, data and the start address, not to debug information. Specifying a non-zero offset is not recommended for an ELF/DWARF file. If the offset causes an address to underflow or overflow an error occurs.

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**.
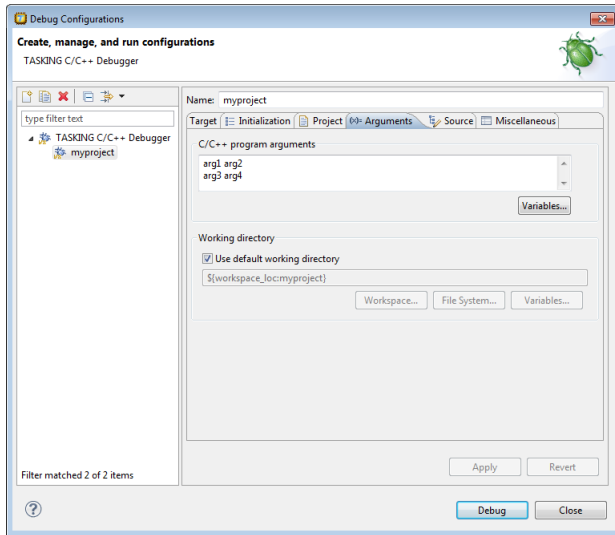
Source tab

On the **Source** tab, you can add additional source code locations in which the debugger should search for debug data.
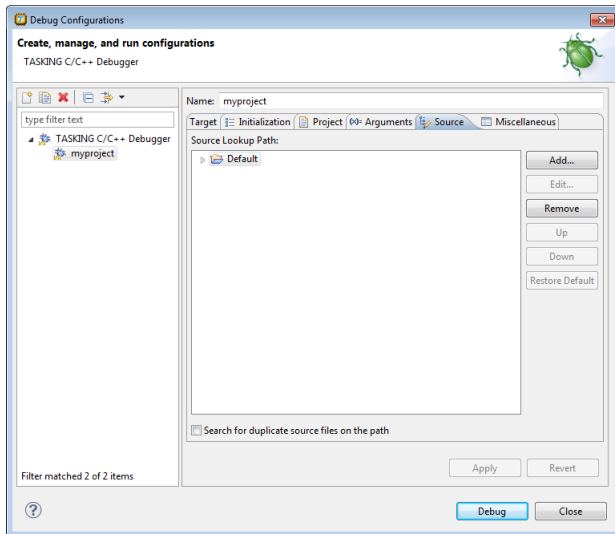
• Usually, the default source code location is correct.

Miscellaneous tab

On the **Miscellaneous** tab you can specify several file locations.
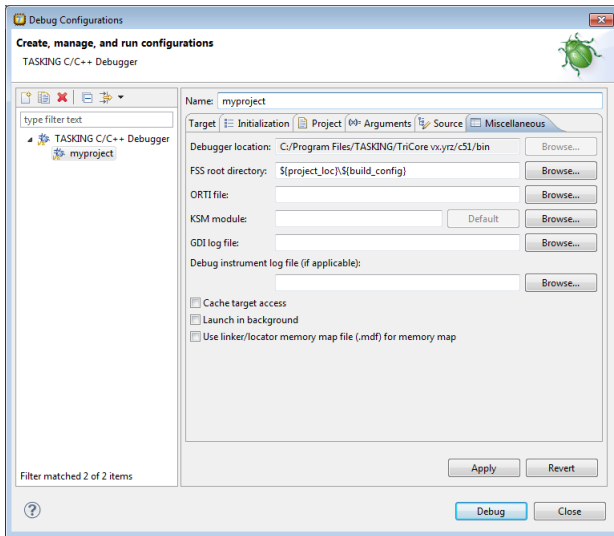
- **Debugger location**

  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. 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.

8.4. 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.
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.

8.5. 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:
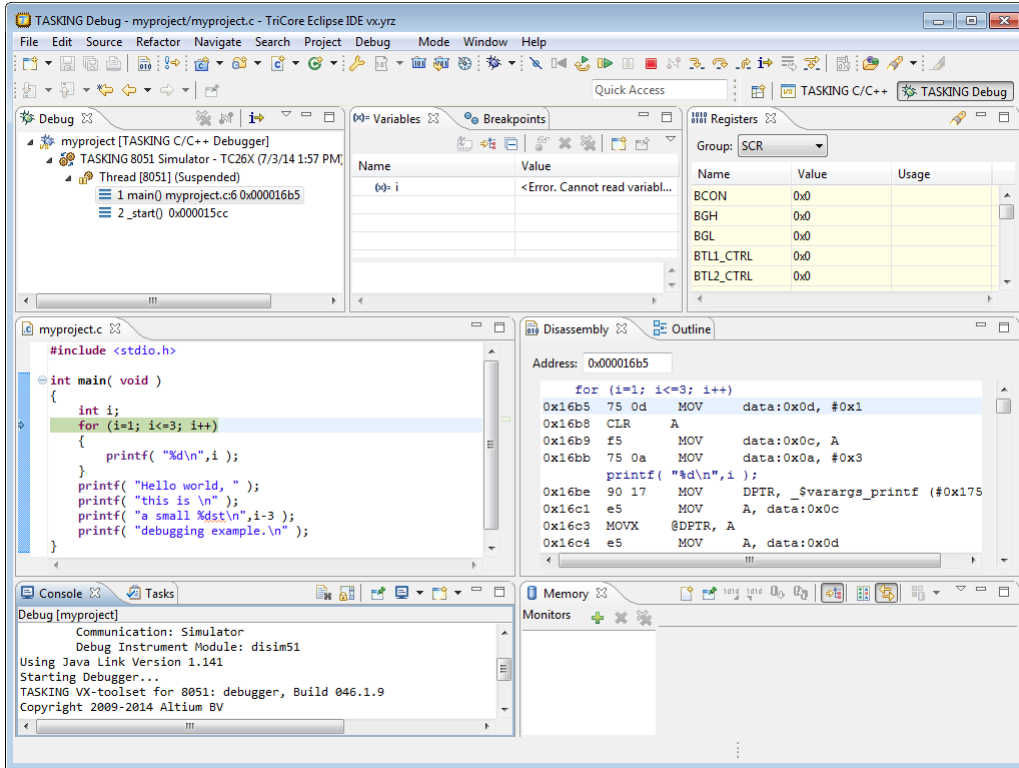
8.5.1. Debug View

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.
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.

8.5.2. 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.

8.5.3. 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 <8051 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 8.3, *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 11.1.3, *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.

8.5.4. Disassembly View

The Disassembly view shows target memory disassembled into instructions and / or data. If possible, the associated 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.

8.5.5. 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.

8.5.6. Memory View

Use the Memory view to inspect and change process memory. The Memory view supports the same addressing as the C language. 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.

8.5.7. 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.

8.5.8. 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.

8.5.9. 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.

8.5.10. 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). 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 8.3, *Creating a Customized Debug Configuration*):
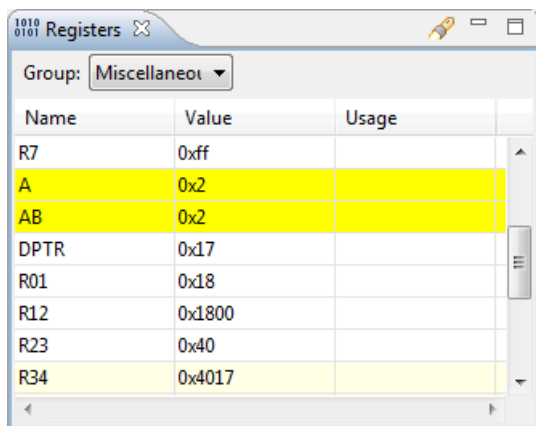
1. From the **Debug** menu, select **Debug Configurations...**

   The Debug Configurations dialog appears.

2. In the left pane, select the configuration you want to change, for example, **TASKING C/C++ Debugger » myproject**.

   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. In the **KSM module** field, specify the name of a KSM shared library file suitable for RTOS kernels.

The debugger supports ORTI specifications v2.0 and v2.1.

8.5.11. 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.

- 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.

8.5.12. 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.

Chapter 9. Tool Options

This chapter provides a detailed description of the options for the compiler, assembler, linker, control program, make utility and the archiver.

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 Compiler

Eclipse option	Description or option
Preprocessing	
Automatic inclusion of '.sfr' file	Control program option **--include-sfr-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**
Memory Model	
Compiler memory model	C compiler option **--model**
Allow reentrant functions	C compiler option **--reentrant**
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**
Treat enumerated types always as integer	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**
Floating-Point	
Floating-point model	Control program option **--fp-model**
Allocation	
Rename sections	C compiler option **--rename-sections**
Clear non-initialized global and static variables	C compiler option **--no-clear**
String allocation	C compiler option **--romstrings**
Default register bank	C compiler option **--registerbank**
Amount of data for automatics	C compiler option **--extend**
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**

Eclipse option	Description or option
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**
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**
MISRA C	
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**
Perform global type checking on C code	C compiler option **--global-type-checking**
Maximum number of emitted errors	C compiler option **--error-limit**
Miscellaneous	
Merge C source code with generated assembly	C compiler option **--source**
Additional options	C compiler options, Control program options

Assembler

Eclipse option	Description or option
Preprocessing	
Use TASKING preprocessor	Assembler option **--preprocessor-type**
Automatic inclusion of '.sfr' file	Control program option **--asm-sfr-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
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	

Eclipse option	Description or option
Data objects	Linker option **--import-object**
Script File	
Defined symbols	Linker option **--define**
Linker script file (.lsl)	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**
Additional options	Linker options

9.1. 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
AS51INC	With this variable you specify one or more additional directories in which the assembler looks for include files. See Section 5.3, *How the Assembler Searches Include Files*.

Environment variable	Description
C51INC	With this variable you specify one or more additional directories in which the C compiler looks for include files. See Section 3.4, *How the Compiler Searches Include Files*.
CC51BIN	When this variable is set, the control program prepends the directory specified by this variable to the names of the tools invoked.
LIBC51	With this variable you specify one or more additional directories in which the linker looks for libraries. See Section 6.3.1, *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.

9.2. 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 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:

```
c51 -Oac test.c
c51 --optimize=+coalesce,+cse test.c
```

When you do not specify an option, a default value may become active.

C compiler option: --cache

Menu entry

1.   Select **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.

You can also enable the cache and specify the cache directory with the environment variable C51CACHE. This option takes precedence over the environment variable.

The cache directory may be shared, for instance by placing it on a network drive.

The compiler creates a directory c51cache in the directory specified with the option **--cache** or the environment variable C51CACHE. 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:

```
c51 --cache=.cache test.c
```

Related information

Section 10.4, *Compiler Cache*

Section 7.6, *Expire Cache Utility*

C compiler option: --cert

Menu entry

1. Select **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 15, *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:

```
c51 --cert=str30 test.c
```

Related information

Chapter 15, *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: --compact-max-size

Menu entry

1. Select **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:

```
c51 --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 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

```
c51 --control-flow-info test.c
```

Related information

Section 7.5.2, *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.

Command line syntax

--core=*core*

You can specify the following *core* arguments:

c51	Generic 8051 architecture
scr2g	XC800 architecture for the AURIX 2G SCR

Default: **c51**

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 option **--cpu**, the 8051 toolset derives the core from the processor you selected. When the default core (**c51**) is used, the macro __CORE_C51__ is defined in the C source file.

With **--core=scr2g**, the compiler generates assembly for the XC800 architecture for the AURIX 2G SCR. The macro __CORE_SCR2G__ is defined in the C source file.

Example

Specify a custom core:

```
c51 --core=scr2g test.c
```

Related information

Control program option **--cpu** (Select processor)

C compiler option: --debug-info (-g)

Menu entry

1. Select **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 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: --define (-D)

Menu entry

1.   Select **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:

```
c51 --define=DEMO test.c
c51 --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.

```
c51 --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

```
c51 --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:

```
c51 --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:

```
c51 --diag=html:all > cerrors.html
```

Related information

Section 3.8, *C Compiler Error Messages*

C compiler option **--cert** (Enable individual CERT checks)

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:

```
c51 --error-file=errors.err test.c
```

Related information

-

C compiler option: --error-limit

Menu entry

1. Select **C Compiler » 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 limit the number of error messages in one compiler run to the specified number. When the limit is exceeded, the compiler aborts with fatal error message F105. Warnings and informational messages are not included in the count. When 0 (zero) or a negative number is specified, the compiler emits all errors. Without this option the maximum number of errors is 42.

Related information

Section 3.8, *C Compiler Error Messages*

C compiler option: --extend

Menu entry

1. Select **C Compiler » Allocation**.

2. Enter the number of bytes in the **Amount of data for automatics** field.

Command line syntax

`--extend=`*value*

Default: 4

Description

By default the compiler uses a maximum of four bytes of internal RAM for pseudo registers. With this option you can change the number of bytes the compiler uses as a maximum.

Related information

Section 1.8.1, *Automatic Variables*

C compiler option: --fp-model

Menu entry

1. Select **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
+/-fastlib	**l/L**	allow less precise library functions
+/-nonan	**n/N**	allow optimizations to ignore NaN/Inf
+/-rewrite	**r/R**	allow expression rewriting
+/-trap	**t/T**	support trapping on exceptions
+/-negzero	**z/Z**	ignore sign of -0.0
	0	alias for **--fp-model=CLNRtZ** (strict)
	1	alias for **--fp-model=cLNRTZ** (precise)
	2	alias for **--fp-model=clnrTz** (fast double)
	3	alias for **--fp-model=clnrTz** (fast single)

Default: **--fp-model=clnrTz**

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=+fastlib** you allow the compiler to select faster but less accurate library functions for certain floating-point operations.

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=+trap** operations trap on floating-point exceptions. 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 1.6, *Pragmas to Control the Compiler*.

C compiler option: --global-type-checking

Menu entry

1. Select **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:

```
c51 -?
c51 --help
c51
```

The following invocation displays a list of the available pragmas:

```
c51 --help=pragmas
```

Related information

-

C compiler option: --include-directory (-I)

Menu entry

1.   Select **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 `C51INC` 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:

```
c51 --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 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

c51 --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: --inline

Menu entry

1. Select **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:

```
c51 --optimize=+compact --inline test.c
```

Related information

C compiler option **--optimize=+compact** (Optimization: code compaction)

Section 1.11.4, *Inlining Functions: inline*

C compiler option: --inline-max-incr / --inline-max-size

Menu entry

1. Select **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

```
c51 --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 1.11.4, *Inlining Functions: inline*
Section 3.6.3, *Optimize for Code Size or Execution Speed*

C compiler option: --integer-enumeration

Menu entry

1. Select **C Compiler » Language**.

2. Enable the option **Treat enumerated types always as integer**.

Command line syntax

```
--integer-enumeration
```

Description

Normally the compiler treats enumerated types as the smallest data type possible (`char` or even `__bit` 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 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:

```
c51 --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

```
c51 --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 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 8051 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;
int main( void )
```

```
{
    p = "hello"; // with -AX the compiler issues warning W525
    return 0;
}
```

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:
    .using 0

    clr    A
    mov    _access+1,A     ; <== Volatile access
    mov    _access,A
    mov    A,R7
    orl    A,_variable+1
    mov    R1,A
    mov    A,R6
    orl    A,_variable
    mov    R0,A
    gjne   R1,#3,_2
    gjne   R0,#0,_2
    clr    A
    mov    R1,A
    mov    R0,A
_2:
    mov    A,R1
    mov    A,R0
```

```
    orl    A,#128
    mov    R0,A
    mov    _access+1,#1    ; <== Volatile access
    mov    _access,#0
    mov    _variable+1,R1  ; <== Moved across volatile access
    mov    _variable,R0
    ret
```

Result with **--language=+volatile**:

```
_TestFunc:
    .using 0

    clr    A
    mov    _access+1,A     ; <== Volatile access
    mov    _access,A
    mov    A,R7
    orl    A,_variable+1
    mov    R1,A
    mov    A,R6
    orl    A,_variable
    mov    R0,A
    gjne   R1,#3,_2
    gjne   R0,#0,_2
    clr    A
    mov    R1,A
    mov    R0,A
_2:
    mov    A,R1
    mov    A,R0
    orl    A,#128
    mov    R0,A
    orl    _variable+1,R1  ; <== Not moved
    orl    _variable,R0
    mov    _access+1,#1    ; <== Volatile access
    mov    _access,#0
    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 1.3, *Shift JIS Kanji Support*

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 `.obj`.

Example

```
c51 --preprocess=+make --make-target=../mytarget.obj test.c
```

The compiler generates dependency lines with the default target name `../mytarget.obj` instead of `test.obj`.

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 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

Menu entry

-

Command line syntax

`--mil`

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.

Related information

Section 3.1, *Compilation Process*

C compiler option: --misrac

Menu entry

1. Select **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

```
c51 --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 3.7.2, *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 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 3.7.2, *C Code Checking: MISRA C*

C compiler option **--misrac**

Linker option **--misrac-report**

C compiler option: --misrac-version

Menu entry

1. Select **C 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 3.7.2, *C Code Checking: MISRA C*

C compiler option **--misrac**

C compiler option: --model (-M)

Menu entry

1. Select **C Compiler » Memory Model**.

2. Select the **Small**, **Auxiliary** or **Large** compiler memory model.

Command line syntax

`--model={small|aux|large}`

`-M{s|a|l}`

Default: `--model=small`

Description

By default, the 8051 compiler uses the small memory model. You can specify the option **--model** to specify another memory model.

The table below illustrates the meaning of each memory model:

Model	Memory type	Location	Pointer size	Pointer arithmetic
small	`__data`	Direct addressable internal RAM	8-bit	8-bit
aux	`__pdata`	One page of external RAM	8-bit	8-bit
large	`__xdata`	External RAM	16-bit	16-bit

The value of the predefined preprocessor symbol __MODEL__ represents the memory model selected with this option. This can be very helpful in making conditional C code in one source module, used for different applications in different memory models. The value of __MODEL__ is:

small model	's'
auxiliary page model	'a'
large model	'l'

Example

To compile the file `test.c` for the large memory model:

```
c51 --model=large test.c
```

Related information

C compiler option **--reentrant** (Enable reentrancy)

Section 1.2.2, *Memory Models*

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 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 3.4, *How the Compiler Searches Include Files*

C compiler option: --no-vector

Menu entry

1. Select **C Compiler » Miscellaneous**.

2. Add the option **--no-vector** to the **Additional options** field.

Command line syntax

```
--no-vector
```

Description

With this option you tell the compiler not to generate code for interrupt vectors and references to the interrupt handler in the run-time library. Use this option if you do not use interrupts in your application, or if you want to write your own interrupt vectors.

Related information

C compiler option **--vector-offset** (Specify a base address for interrupt vectors)

Section 1.11.5, *Interrupt Functions*

C compiler option: --no-warnings (-w)

Menu entry

1.   Select **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:

```
c51 test.c --no-warnings=537,538
```

Related information

C compiler option **--warnings-as-errors** (Treat warnings as errors)

Pragma `warning`

C compiler option: --optimize (-O)

Menu entry

1.   Select **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
+/-ipro	**b/B**	Interprocedural register optimizations
+/-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
+/-loop	**l/L**	Loop transformations
+/-forward	**o/O**	Forward store
+/-propagate	**p/P**	Constant propagation
+/-compact	**r/R**	Code compaction (reverse inlining)
+/-subscript	**s/S**	Subscript strength reduction
+/-peephole	**y/Y**	Peephole optimizations

Use the following options for predefined sets of flags:

--optimize=0	**-O0**	No optimization
		Alias for **-OaBCEFGILOPRSY**

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 **-OabcefgILOPRSy**

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 **-Oabcefglloprsy**

Enables more optimizations to reduce code size and/or execution time. This is the default optimization level.

--optimize=3	**-O3**	Optimize most
		Alias for **-Oabcefgiloprsy**

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:

```
c51 test.c

c51 --optimize=2 test.c
c51 -O2 test.c

c51 --optimize test.c
c51 -O test.c

c51 -OabcefgIloprsy test.c
c51 --optimize=+coalesce,+ipro,+cse,+expression,+flow,+glo,
     -inline,+loop,+forward,+propagate,+compact,
     +subscript,+peephole test.c
```

Related information

C compiler option **--tradeoff** (Trade off between speed and size)

Pragma `optimize/endoptimize`

Section 3.6, *Compiler Optimizations*

C compiler option: --option-file (-f)

Menu entry

1. Select **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:

```
c51 --option-file=myoptions
```

This is equivalent to the following command line:

```
c51 --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:

```
c51 --output=output.src test.c
```

Related information

-

C compiler option: --preprocess (-E)

Menu entry

1. Select **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 .obj. 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

```
c51 --preprocess=+comments,+includes,-list,-make,-noline test.c --output=test.pre
```

The compiler preprocesses the file `test.c` and sends the output to the file `test.pre`. Comments and a list of all included source files are included but no list of macro definitions and 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 Compiler » Debugging**.

2. Enable or disable **Static profiling**.

Command line syntax

--profile[**=**_flag_,...]

-p[_flags_]

You can set the following flags:

+/-static	**s/S**	static profile generation

Default: **-ps**

Default (**-p** without flags): **-ps**

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.

For an extensive description of profiling refer to Chapter 4, *Profiling*.

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 option **Generate symbolic debug information** (**--debug-info**) does not affect profiling, execution time or code size.

Example

To generate static profiling information for the module `test.c`, compile as follows:

```
c51 --profile=+static test.c
```

Related information

Chapter 4, *Profiling*

C compiler option: --reentrant

Menu entry

1. Select **C Compiler » Memory Model**.

2. Enable the option **Allow reentrant functions**.

Command line syntax

```
--reentrant
```

Description

If you select reentrancy, a (less efficient) virtual dynamic stack is used which allows you to call functions recursively. With reentrancy, you can call functions at any time, even from interrupt functions.

Related information

C compiler option **--model** (Memory model)

Section 1.2.2, *Memory Models*

C compiler option: --registerbank

Menu entry

1. Select **C Compiler » Allocation**.

2. In the **Default register bank** field, select **0**, **1**, **2**, **3** or **register bank independent**.

Command line syntax

`--registerbank={0 | 1 | 2 | 3 | n | none}`

Description

With this option you select the default register bank. For normal functions no code is generated to switch to the register bank. This will only be done for interrupt functions. When you select **none** (**n**) the generated code will be register bank independent and a switch will never be generated.

Related information

Section 1.11.5.2, *Register Bank Switching: __bankx / __nobank*

C compiler option: --relax-compact-name-check

Menu entry

1.   Select **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. Chunks of code that are part of functions with a different 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 3.6.2, *Core Specific Optimizations (backend)*

C compiler option: --relax-overlay-name-check

Menu entry

1. Select **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 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 Compiler » Allocation**

    The Rename sections box shows the sections that are currently renamed.

2. To rename a section, click on the **Add** button in the **Rename sections** box.

3. Type the rename rule in the format *type=format* or *format* (for example, `data={module}_{attrib}`)

Use the **Edit** and **Delete** button to change a section renaming or to remove an entry from the list.

Command line syntax

`--rename-sections=`[*type*`=`]*format_string*`[,`[*type*`=`]*format_string*`]...`

`-R`[*type*`=`]*format_string*`[,`[*type*`=`]*format_string*`]...`

Default section name: {type}_{name}

Description

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. You can then use this unique section name in the linker script file for locating.

With the memory *type* you select which sections are renamed. The matching sections will get the specified *format_string* for the section name. The format string can contain characters and may contain the following format specifiers:

`{attrib}`	section attributes, separated by underscores
`{module}`	module name
`{name}`	object name, name of variable or function
`{type}`	section type

Instead of this option you can also uses the pragmas `section`/`endsection` in the C source.

Example

To rename sections of memory type `data` to _c51_test_*variable_name*:

```
c51 --rename-sections=data=_c51_{module}_{name} test.c
```

Related information

See assembler directive `.SEGMENT` for a list of section types and attributes.

Pragmas `section/endsection`

Section 1.12, *Section Naming*

C compiler option: --romstrings

Menu entry

1. Select **C Compiler » Allocation**.

2. In the **String allocation** field, select **Keep strings in ROM (use __rom keyword for a pointer to a string)**.

Command line syntax

```
--romstrings
```

Description

By default, constant strings are copied from ROM to RAM at program startup. With this option you tell the compiler to keep constant strings in ROM. If you use this option, you can access these strings only with the `__rom` keyword.

With this option enabled, strings are *not* copied to RAM at startup to save RAM memory. Strings in ROM cannot be modified and access is slower than access to strings in RAM.

Related information

Pragma `ramstring`/`romstring`

Section 1.9, *Strings*

C compiler option: --signed-bitfields

Menu entry

1. Select **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

Section 1.1, *Data Types*

C compiler option: --source (-s)

Menu entry

1.   Select **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: --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: --tradeoff (-t)

Menu entry

1. Select **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.

Example

To set the trade-off level for the used optimizations:

```
c51 --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 3.6.3, *Optimize for Code Size or Execution Speed*

C compiler option: --uchar (-u)

Menu entry

1. Select **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

Section 1.1, *Data Types*

C compiler option: --undefine (-U)

Menu entry

1. Select **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__`:

```
c51 --undefine=__TASKING__ test.c
```

Related information

C compiler option **--define** (Define preprocessor macro)

Section 1.7, *Predefined Preprocessor Macros*

C compiler option: --vector-offset

Menu entry

1.   Select **C Compiler » Miscellaneous**.

2.   Add the option **--vector-offset** to the **Additional options** field.

Command line syntax

--vector-offset=*value*

Default: **--vector-offset=0**

Description

With this option you can specify a 16-bit offset address for the interrupt vector table. The default offset address is 0x0000.

This option is for example useful to place the vector table in RAM.

Example

Specify 0x4000 as the base address of the interrupt vector table on the command line:

```
c51 --vector-offset=0x4000 test.c
```

Suppose your C source contains the following interrupt function:

```
__interrupt(0x0013) void isr( void )
```

The compiler adds the offset address to the vector address in the C source so the actual vector address becomes 0x4013.

Related information

C compiler option **--no-vector** (Do not generate interrupt vectors)

Section 1.11.5, *Interrupt Functions*

C compiler option: --verbose (-v)

Menu entry

-

Command line syntax

`--verbose`

`-v`

Description

With this option you put the C compiler in verbose mode. The C compiler performs its tasks while it prints the steps it performs to stdout.

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.

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

9.3. 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:

```
as51 -Ogs test.src
as51 --optimize=+generics,+instr-size test.src
```

When you do not specify an option, a default value may become active.

Assembler option: --allow-undefined-macro

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--allow-undefined-macro** to the **Additional options** field.

Command line syntax

```
--allow-undefined-macro
```

Description

With this option the macro preprocessor part of the assembler will not issue an error when it finds an undefined macro.

Related information

-

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`.

```
as51 --case-insensitive test.src
```

Related information

-

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: --control

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--control** to the **Additional options** field.

Command line syntax

--control=*control*

Description

With this option you can specify an assembler control on the command line. Use the name of the control without the '$' prefix. You can use this option multiple times.

Example

```
as51 --control="message(This is a control on the command line)" test.src
```

```
This is a control on the command line
```

Related information

Section 2.9.2, *Assembler Controls*

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:

c51	Generic 8051 architecture
scr2g	XC800 architecture for the AURIX 2G SCR

Default: **c51**

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 option **--cpu**, the 8051 toolset derives the core from the processor you selected. When the default core (**c51**) is used, the define __CORE_C51__ is set to 1.

With **--core=scr2g**, you can use predefined XC800 SFRs for the AURIX 2G SCR in the assembly code. The define __CORE_SCR2G__ is set to 1.

Example

To allow the use of predefined XC800 SFRs for the AURIX 2G SCR in the assembly code, enter:

```
as51 --core=scr2g test.src
```

Related information

Control program option **--cpu** (Select processor)

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:

```
as51 --define=DEMO test.src
as51 --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

```
as51 --dep-file=test.dep test.src
```

The assembler assembles the file `test.src`, which results in the output file `test.obj`, 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:

```
as51 --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:

```
as51 --diag=html:all > aserrors.html
```

Related information

Section 5.6, *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 input file with extension `.ers`.

Example

To write errors to `errors.ers` instead of `stderr`, enter:

```
as51 --error-file=errors.ers test.src
```

Related information

Section 5.6, *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 5.6, *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:

```
as51 -?
as51 --help
as51
```

To see a detailed description of the available options, enter:

```
as51 --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 `AS51INC` 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:

```
as51 --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

```
as51 --include-file=myinc.inc test.src
```

The file `myinc.inc` 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: --info-messages

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--info-messages** to the **Additional options** field.

Command line syntax

```
--info-messages
```

Description

With this option the macro preprocessor can generate informational messages in addition to errors or warnings.

Related information

-

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 (`.obj`) 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 source 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 (`.SECTION`)
+/-symbol	**e/E**	List symbol definition directives
+/-generic-expansion	**g/G**	List expansion of generic instructions
+/-generic	**i/I**	List generic instructions
+/-line	**l/L**	List #line directives
+/-empty-line	**n/N**	List empty source lines and comment lines (newline)
+/-equate	**q/Q**	List equate and set directives (`.EQU`, `.SET`)
+/-relocations	**r/R**	List relocations characters 'r'
+/-hll	**s/S**	List HLL symbolic debug informations
+/-equate-values	**v/V**	List equate and set values
+/-wrap-lines	**w/W**	Wrap source lines
+/-cycle-count	**y/Y**	List cycle counts

Use the following options for predefined sets of flags:

--list-format=0	**-L0**	All options disabled Alias for **--list-format=DEGILNQRSVWY**
--list-format=1	**-L1**	All options enabled Alias for **--list-format=degilnqrsvwy**

Default: **--list-format=dEGilnqrsVwy**

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 `.obj`.

Example

```
as51 --dep-file --make-target=../mytarget.obj test.src
```

The assembler generates dependency lines with the default target name `../mytarget.obj` instead of `test.obj`.

Related information

Assembler option **--dep-file** (Generate dependencies in a file)

Assembler option: --max-nesting

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--max-nesting** to the **Additional options** field.

Command line syntax

--max-nesting=*number*

Default: 31

Description

With this option you can set the maximum include file nesting level.

Related information

%INCLUDE()

Assembler option: --no-skip-asm-comment

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--no-skip-asm-comment** to the **Additional options** field.

Command line syntax

`--no-skip-asm-comment`

Description

With this option you instruct the macro preprocessor not to skip parsing after assembly comment ';'.

Related information

-

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:

```
as51 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 5.4, *Assembler Optimizations*

When you use this option to specify a set of optimizations, you can turn on or off the optimizations in your
assembly source file with the assembler controls $optimize/$nooptimize.

Related information

Assembler control **$OPTIMIZE**

Section 5.4, *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-info=+asm,-local
test.src
```

Specify the option file to the assembler:

```
as51 --option-file=myoptions
```

This is equivalent to the following command line:

```
as51 --debug-info=+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 `.obj`.

Example

To create the file `relobj.obj` instead of `asm.obj`, enter:

```
as51 --output=relobj.obj 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 **$PAGELENGTH**

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 **$PAGEWIDTH**

Assembler option: --parameters-redefine

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--parameters-redefine** to the **Additional options** field.

Command line syntax

`--parameters-redefine`

Description

With this option it is allowed to use the `%SET` macro to redefine a macro parameter.

Related information

`%SET()`

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

1. Select **Assembler » Preprocessing**.

2. Enable or disable the option **Use TASKING preprocessor**.

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: --prompt

Menu entry

1.   Select **Assembler » Miscellaneous**.

2.   Add the option **--prompt** to the **Additional options** field.

Command line syntax

--prompt=`string`

Default: >

Description

With this option you can set the prompt for the `%IN` built-in function.

Example

To set the prompt for the `%IN` function to "`cmd>`", enter:

```
as51 --prompt="cmd>" test.src
```

Related information

`%IN()`

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:

```
as51 --list-file --section-info asm.src
```

Related information

Assembler option **--list-file** (Generate list file)

Assembler option: ---sfr-file

Menu entry

1.   Select **Assembler » Preprocessing**.

2.   Enable the option **Automatic inclusion of '.sfr' file**.

Command line syntax

--sfr-file=reg*cpu***.sfr**

Description

With this option the assembler can include the register file reg*cpu*.sfr.

In Eclipse this option is called automatically by default.

Related information

Section 2.6, *Special Function Registers*

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 **.PUBLIC**

Assembler option: --version (-V)

Menu entry

-

Command line syntax

```
--version
```

```
-V
```

Description

Display version information. The assembler ignores all other options or input files.

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)

Assembler option: --warn-on-undefined-macro

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--warn-on-undefined-macro** to the **Additional options** field.

Command line syntax

```
--warn-on-undefined-macro
```

Description

With this option the assembler generates warning W 201 instead of error E 301 when an undefined preprocessor macro name is found.

Related information

-

9.4. 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:

```
lk51 -mfkl test.obj
lk51 --map-file-format=+files,+link,+locate test.obj
```

When you do not specify an option, a default value may become active.

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 `.obj` file case insensitive.

Related information

Assembler option **--case-insensitive**

Linker option: --chip-output (-c)

Menu entry

-

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:

```
lk51 --chip-output=myfile:IHEX test1.obj
```

In this case, this generates the file `myfile_`*memname*`.hex`.

Related information

Linker option **--output** (Output file)

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 symbol __CPU__ which is used in the linker script file default.lsl to include the proper processor specific LSL file, enter:

lk51 --define=__CPU__=tc26x test.obj

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:

```
lk51 --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:

```
lk51 --diag=html:all > lkerrors.html
```

Related information

Section 6.10, *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 `lk51.elk`.

Example

To write errors to `errors.elk` instead of `stderr`, enter:

```
lk51 --error-file=errors.elk test.obj
```

Related information

Section 6.10, *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 6.10, *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:

```
lk51 mylib.lib
```

Nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through `mylib.lib`.

```
lk51 --extern=__START mylib.lib
```

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.lib`. This process repeats until no new unresolved symbols are found.

Related information

Section 6.3, *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:

```
lk51 --first-library-first a.lib test.obj b.lib
```

If the file `test.obj` calls a function which is both present in `a.lib` and `b.lib`, normally the function in `b.lib` would be extracted. With this option the linker first tries to extract the symbol from the first library `a.lib`.

Note that routines in `b.lib` that call other routines that are present in both `a.lib` and `b.lib` are now also resolved from `a.lib`.

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 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:

```
lk51 -?
lk51 --help
lk51
```

To see a detailed description of the available options, enter:

```
lk51 --help=options
```

Related information

-

Linker option: --hex-format

Menu entry

1.   Select **Linker » Miscellaneous**.

2.   Add the option **--hex-format** to the **Additional options** field.

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)

Linker option: --hex-record-size

Menu entry

1. Select **Linker » Miscellaneous**.

2. 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 13.2, *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 6.5, *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:

```
lk51 --include-directory=c:\proj\include --lsl-file=mylsl.lsl test.obj
```

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.obj`, `test2.obj` and `test3.obj` are incrementally linked:

1. `lk51 --incremental test1.obj test2.obj --output=test.out`

   test1.obj and test2.obj are linked

2. `lk51 --incremental test3.obj test.out`

   test3.obj and test.out are linked, task1.out is created

3. `lk51 task1.out`

   task1.out is located

Related information

Section 6.4, *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 *name*`.lib`, 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 variable `LIBC51`, unless you used the option **--ignore-default-library-path**.

Example

To search in the system library `c51ss0.lib` (C library):

```
lk51 test.obj mylib.lib --library=c51ss0
```

The linker links the file `test.obj` and first looks in library `mylib.lib` (in the current directory only), then in the system library `c51ss0.lib` to resolve unresolved symbols.

Related information

Linker option **--library-directory** (Additional search path for system libraries)

Section 6.3, *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.

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 variable LIBC51. 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 variable LIBC51.

3. The default directory $(PRODDIR)\lib.

Example

Suppose you call the linker as follows:

```
lk51 test.obj --library-directory=c:\mylibs --library=c51ss0
```

First the linker looks in the directory `c:\mylibs` for library `c51ss0.lib` (this option). If it does not find the requested libraries, it looks in the directory that is set with the environment variable `LIBC51`. Then the linker looks in the default directory `$(PRODDIR)\lib` for libraries.

Related information

Linker option **--library** (Link system library)

Section 6.3.1, *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 6.7, *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 `lk51.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 TriCore 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.

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 `default.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 6.7, *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 (`.obj`) 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 12.2, *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_,...

-m_flags_

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 12.2, *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: --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.

Related information

-

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: --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:

```
lk51 --no-warnings=135,136 test.obj
```

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*`,...`

`-O`*flags*

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.

Note that when you use the flag **+copytable-compression**, sections affected by the copy table are located as if they were in a clustered LSL group, if they do not have a locate restriction yet.

Related information

For details about each optimization see Section 6.6, *Linker Optimizations*.

Define the mutual order of sections in an LSL group in Section 14.8.2, *Creating and Locating Groups of Sections*.

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.obj                       (input file)
--library-directory=c:\mylibs  (additional search path for system libraries)
```

Specify the option file to the linker:

```
lk51 --option-file=myoptions
```

This is equivalent to the following command line:

```
lk51 --map-file=my.map test.obj --library-directory=c:\mylibs
```

Related information

-

Linker option: --output (-o)

Menu entry

-

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 `near`, enter:

`lk51 test.obj --output=myfile.hex:IHEX:2,near`

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: --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

-

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.

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)

9.5. Control Program Options

The control program **cc51** facilitates the invocation of the various components of the 8051 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, assembler or linker, it is recommended to use the control program options **--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:

```
cc51 -Wc-Oac test.c
cc51 --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

-

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:

```
cc51 --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: --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: --control-flow-info

Menu entry

1. Select **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

```
cc51 --control-flow-info test.c
```

Related information

Section 7.5.2, *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.

Command line syntax

--core=_core_

You can specify the following _core_ arguments:

c51	Generic 8051 architecture
scr2g	XC800 architecture for the AURIX 2G SCR

Default: **c51**

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 option **--cpu**, the 8051 toolset derives the core from the processor you selected. When the default core (**c51**) is used, the macro __CORE_C51__ is defined in the C source file.

With **--core=scr2g**, the compiler generates assembly for the XC800 architecture for the AURIX 2G SCR. The macro __CORE_SCR2G__ is defined in the C source file.

Example

To select the XC800 architecture for the AURIX 2G SCR, enter:

```
cc51 --core=scr2g test.c
```

Related information

Control program option **--cpu** (Select processor)

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 TC26X, or a base *CPU* name, like tc26x or its unique *id*, like tc26x.

Based on this option the C compiler and assembler can include the special function register file reg*cpu*.sfr, if you also specify option **--include-sfr-file** or option **--asm-sfr-file**. In Eclipse this is done automatically.

The standard list of supported processors is defined in the file processors.xml. This file defines for each processor its full name (for example, TC26X), its ID, the base CPU name (for example, tc26x), the core settings (for example, 51) and the on-chip flash settings 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 tc26x)

2. if none matched, match with the 'name' attribute in processors.xml (case insensitive, for example TC26X)

3. if still none matched, match any of the base CPU names (the 'cpu' attribute in processors.xml, for example tc26x). 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=tc26x**. 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__=tc26x** to the linker. If you also specify option **--include-sfr-file**, the control program passes the option **-Hsfr/regtc26x.sfr** to the C compiler. If you also specify option **--asm-sfr-file**, the control program passes the option **--sfr-file=sfr/regtc26x.sfr** to the assembler.

Example

To generate the file test.elf for the TC26X processor, enter:

```
cc51 --cpu=tc26x --include-sfr-file --asm-sfr-file -v -t test.c
```

The control program will call the tools as follows:

```
+ c51  -Ms --registerbank=0 -Hsfr/regtc26x.sfr -o test.src test.c
+ as51 --sfr-file=sfr/regtc26x.sfr -o test.obj test.src
+ lk51 -o test.elf -D__CPU__=tc26x --map-file test.obj
       -lc51ss0 -lfp51ss -lrt51
```

Related information

Control program option **--cpu-list** (Show list of processors)

Control program option **--processors** (Read additional processor definitions)

Control program option **--include-sfr-file** (Include SFR file in compiler)

Control program option **--asm-sfr-file** (Include SFR file in assembler)

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:

```
cc51 --cpu-list
```

To show all processors that have tc26 in their name, enter:

```
cc51 --cpu-list=tc26

--- ~/c51/etc/processors.xml ---
    id          name        CPU         core
    tc26x       TC26x       tc26x       c51
    tc26xb      TC26xB      tc26xb      c51
```

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:

relocatable	**l**	Stop after the files are linked to a linker object file (.out)
mil	**m**	Stop after C files are compiled to MIL (.mil)
object	**o**	Stop after the files are assembled to objects (.obj)
assembly	**s**	Stop after 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.obj:

```
cc51 --create test.c
```

The control program stops after the file is assembled. It does not link nor locate the generated output.

Related information

Linker option **--link-only** (Link only, no locating)

Control program option: --debug-info (-g)

Menu entry

1. Select **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 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:

```
cc51 --define=DEMO test.c
cc51 --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.

```
cc51 --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)

437

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

```
cc51 --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** | *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 103, enter:

```
cc51 --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:

```
cc51 --diag=html:all > ccerrors.html
```

Related information

Section 3.8, *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.

Example

To write errors to error files instead of stderr, enter:

```
cc51 --error-file test.c
```

Related information

Control Program option **--warnings-as-errors** (Treat warnings as errors)

Control program option: --error-limit

Menu entry

1. Select **C Compiler » 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 limit the number of error messages in one invocation to the specified number. When the limit is exceeded, the control program aborts with fatal error message F105. Warnings and informational messages are not included in the count. When 0 (zero) or a negative number is specified, the control program emits all errors. Without this option the maximum number of errors is 42. The control program also passes this option to the C compiler, assembler and linker.

Related information

Section 3.8, *C Compiler Error Messages*

Control program option: --format

Menu entry

-

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:

```
cc51 --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 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
+/-fastlib	**l/L**	allow less precise library functions
+/-nonan	**n/N**	allow optimizations to ignore NaN/Inf
+/-rewrite	**r/R**	allow expression rewriting
+/-trap	**t/T**	support trapping on exceptions
+/-negzero	**z/Z**	ignore sign of -0.0
	0	alias for **--fp-model=CLNRtZ** (strict)
	1	alias for **--fp-model=cLNRTZ** (precise)
	2	alias for **--fp-model=clnrTz** (fast double)
	3	alias for **--fp-model=clnrTz** (fast single)

Default: **`--fp-model=clnrTz`**

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=+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=+trap** operations trap on floating-point exceptions. By default the control program uses the non-trapping floating-point library (`fp51ss.lib`). With this option you tell the control program to use the trapping floating-point library (`fp51sst.lib`).

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 1.6, *Pragmas to Control the Compiler*.

Control program option: --global-type-checking

Menu entry

1. Select **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:

```
cc51 -?
cc51 --help
cc51
```

To see a detailed description of the available options, enter:

```
cc51 --help=options
```

Related information

-

Control program option: --include-directory (-I)

Menu entry

1. Select **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:

```
cc51 --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-sfr-file / --asm-sfr-file

Menu entry

1. Select **C Compiler » Preprocessing**.

2. Enable the option **Automatic inclusion of '.sfr' file**.

3. Select **Assembler » Preprocessing**.

4. Enable the option **Automatic inclusion of '.sfr' file**.

Command line syntax

```
--include-sfr-file
```

```
--asm-sfr-file
```

Description

With **--include-sfr-file** the compiler includes the register file reg*cpu*.sfr as based on the selected target processor.

With **--asm-sfr-file** the assembler includes the register file reg*cpu*.sfr as based on the selected target processor.

In Eclipse both options are enabled by default.

Example

```
cc51 --cpu=tc26x --include-sfr-file --asm-sfr-file -v -t test.c

+ c51 -Ms --registerbank=0 -Hsfr/regtc26x.sfr -o test.src test.c
+ as51 --sfr-file=sfr/regtc26x.sfr -o test.obj test.src
+ lk51 -o test.elf -D__CPU__=tc26x --map-file test.obj
       -lc51ss0 -lfp51ss -lrt51
```

Related information

Control program option **--cpu** (Select processor)

Section 1.2.5, *Accessing Hardware from C: __sfr, __bsfr*

Control program option: --integer-enumeration

Menu entry

1. Select **C Compiler » Language**.

2. Enable the option **Treat enumerated types always as integer**.

Command line syntax

`--integer-enumeration`

Description

Normally the compiler treats enumerated types as the smallest data type possible (`char` 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: --iso

Menu entry

1.   Select **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:

```
cc51 --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

```
cc51 --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 `.obj` 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

```
cc51 --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 *name*.lib, 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 variable LIBC51, unless you used the option **--ignore-default-library-path**.

Example

To search in the system library c51ss0.lib (C library):

```
cc51 test.obj mylib.lib --library=c51ss0
```

The linker links the file test.obj and first looks in library mylib.lib (in the current directory only), then in the system library c51ss0.lib 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 6.3, *Linking with Libraries*

Chapter 11, *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`.

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 variable `LIBC51`. 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 variable `LIBC51`.

3. The default directory `$(PRODDIR)\lib`.

Example

Suppose you call the control program as follows:

```
cc51 test.c --library-directory=c:\mylibs --library=c51ss0
```

First the linker looks in the directory `c:\mylibs` for library `c51ss0.lib` (this option). If it does not find the requested libraries, it looks in the directory that is set with the environment variable `LIBC51`. Then the linker looks in the default directory `$(PRODDIR)\lib` for libraries.

Related information

Control program option **--library** (Link system library)

Section 6.3.1, *How the Linker Searches Libraries*

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-file (-d)

Menu entry

An LSL file can be generated when you create your TriCore 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.

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 6.7, *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 `.obj`.

Example

```
cc51 --preprocess=+make --make-target=../mytarget.obj test.c
```

The compiler generates dependency lines with the default target name `../mytarget.obj` instead of `test.obj`.

Related information

Control program option **--preprocess=+make** (Generate dependencies for make)

Control program option **--dep-file** (Generate dependencies in a file)

Control program option: --model (-M)

Menu entry

1. Select **C Compiler » Memory Model**.

2. Select the **Small**, **Auxiliary** or **Large** compiler memory model.

Command line syntax

--model={small|aux|large}

-M{s|a|l}

Default: **--model=small**

Description

By default, the 8051 compiler uses the small memory model. You can specify the option **--model** to specify another memory model.

The table below illustrates the meaning of each memory model:

Model	Memory type	Location	Pointer size	Pointer arithmetic
small	__data	Direct addressable internal RAM	8-bit	8-bit
aux	__pdata	One page of external RAM	8-bit	8-bit
large	__xdata	External RAM	16-bit	16-bit

The value of the predefined preprocessor symbol __MODEL__ represents the memory model selected with this option. This can be very helpful in making conditional C code in one source module, used for different applications in different memory models. The value of __MODEL__ is:

small model	's'
auxiliary page model	'a'
large model	'l'

Example

To compile the file `test.c` for the large memory model:

```
cc51 --model=large test.c
```

Related information

Control program option **--reentrant** (Enable reentrancy)

Section 1.2.2, *Memory Models*

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

cc51 --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 (c51ss0.lib) and avoid unresolved externals:

cc51 --no-default-libraries --library=c51ss0 test.c

Related information

Control program option **--library** (Link system library)

Section 6.3.1, *How the Linker Searches Libraries*

Chapter 11, *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 (`.obj`) 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 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:

```
cc51 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:

```
cc51 --option-file=myoptions
```

This is equivalent to the following command line:

```
cc51 —-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

```
cc51 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`:

```
cc51 --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 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 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-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:

```
cc51 --pass-linker=--verbose test.c
```

Related information

-

Control program option: --preprocess (-E) / --no-preprocessing-only

Menu entry

1. Select **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 `.obj`. 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

```
cc51 --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 » 8051**.

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, TC26X), its ID, the base CPU name (for example, tc26x) and if present the on-chip flash settings for that processor. Each processor also defines an option to supply to the linker for preprocessing the LSL file for the applicable on-chip memory definitions.

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:

```
cc51 --processors=new-processors.xml --cpu=NEW-PROC test.c
```

Related information

Control program option **--cpu** (Select processor)

Control program option: --profile (-p)

Menu entry

1. Select **C Compiler » Debugging**.

2. Enable or disable **Static profiling**.

Command line syntax

--profile[**=**_flag_,...]

-p[_flags_]

You can set the following flags:

> **+/-static**   **s/S**   static profile generation

Default: **-ps**

Default (**-p** without flags): **-ps**

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.

For an extensive description of profiling refer to Chapter 4, *Profiling*.

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 option **Generate symbolic debug information** (**--debug-info**) does not affect profiling, execution time or code size.

Example

To generate static profiling information for the module `test.c` during execution, compile as follows:

```
cc51 --profile=+static test.c
```

Related information

Chapter 4, *Profiling*

Control program option: --reentrant

Menu entry

1. Select **C Compiler » Memory Model**.

2. Enable the option **Allow reentrant functions**.

Command line syntax

```
--reentrant
```

Description

If you select reentrancy, a (less efficient) virtual dynamic stack is used which allows you to call functions recursively. With reentrancy, you can call functions at any time, even from interrupt functions.

Related information

Control program option **--model** (Memory model)

Section 1.2.2, *Memory Models*

Control program option: --registerbank

Menu entry

1. Select **C Compiler » Allocation**.

2. In the **Default register bank** field, select **0**, **1**, **2**, **3** or **register bank independent**.

Command line syntax

```
--registerbank={0 | 1 | 2 | 3 | n | none}
```

Description

With this option you select the default register bank. For normal functions no code is generated to switch to the register bank. This will only be done for interrupt functions. When you select **none** (**n**) the generated code will be register bank independent and a switch will never be generated.

Related information

Section 1.11.5.2, *Register Bank Switching: __bankx / __nobank*

Control program option: --signed-bitfields

Menu entry

1. Select **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: --uchar (-u)

Menu entry

1. Select **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

Section 1.1, *Data Types*

Control program option: --undefine (-U)

Menu entry

1. Select **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__`:

```
cc51 --undefine=__TASKING__ test.c
```

Related information

Control program option **--define** (Define preprocessor macro)

Section 1.7, *Predefined Preprocessor Macros*

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 it 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)

9.6. Make Utility Options

When you build a project in Eclipse, Eclipse generates a makefile and uses the make utility **mk51** 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:

mk51 [*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 7.3, *Make Utility mk51*.

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.obj main.obj
             lk51 demo.obj main.obj -lc51ss0 -lfp51ss -lrt51
else
   real.elf : real.obj main.obj
             lk51 real.obj main.obj -lc51ss0 -lfp51ss -lrt51
endif
```

You can now use a macro definition to set the DEMO flag:

```
mk51 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.obj` or with `real.obj`.

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:

```
mk51 -?
```

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

`mk51 -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

```
mk51 -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 **mk51**.

With the option **-DD** not only the lines of the makefile are printed but also the lines of the `mk51.mk` file (implicit rules).

Example

```
mk51 -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

`mk51 -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

```
mk51 -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

```
mk51 -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

```
mk51 -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:

```
mk51 -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

```
mk51 -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

```
mk51 -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

```
mk51 -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:

```
mk51 -m myoptions
```

This is equivalent to the following command line:

```
mk51 -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

`mk51 -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

`mk51 –p`

The make utility never removes target dependency files.

Related information

Special target `.PRECIOUS` in Section 7.3.2.1, *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

```
mk51 -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 `mk51.mk`, then it reads the makefile with the rules to build your files. (The file `mk51.mk` is located in the `\etc` directory of the toolset.)

With this option you tell the make utility not to read `mk51.mk` and to rely fully on the make rules in the makefile.

Example

```
mk51 -r
```

The make utility does not read the implicit make rules in `mk51.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 `mk51.mk` and to rely fully on the make rules in the makefile.

Example

`mk51 -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 **mk51** 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

```
mk51 -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

`mk51 -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

mk51 –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.

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

```
mk51 -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

```
mk51 -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

```
mk51 -x
```

If errors occur, the make utility gives extended information.

Related information

-

9.7. 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 7.2, *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** | *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 169, enter:

```
amk --diag=451
```

This results in the following message and explanation:

```
E451: make stopped

An error has 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[_item_]

-?

You can specify the following arguments:

>   **options**   **o**   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 -h
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 Behavior tab, select **Enable 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)

9.8. Archiver Options

The archiver and library maintainer **ar51** 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:

ar51 *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 7.4, *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: --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

```
ar51 --delete mylib.lib obj1.obj obj2.obj
```

The archiver deletes `obj1.obj` and `obj2.obj` from the library `mylib.lib`.

```
ar51 -d -v mylib.lib obj1.obj obj2.obj
```

The archiver deletes `obj1.obj` and `obj2.obj` from the library `mylib.lib` and displays which files are removed.

Related information

-

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:

```
ar51 --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:

```
ar51 --diag=html:all > arerrors.html
```

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

```
ar51 --dump mylib.lib obj1.obj > file.obj
```

The archiver prints the file `obj1.obj` to standard output where it is redirected to the file `file.obj`. 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.obj` from the library `mylib.lib`:

```
ar51 --extract mylib.lib obj1.obj
```

If you do not specify an object module, all object modules are extracted:

```
ar51 -x mylib.lib
```

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:

```
ar51 -?
ar51 --help
ar51
```

To see a detailed description of the available options, enter:

```
ar51 --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.lib` contains the following objects (see option **--print**):

```
obj1.obj
obj2.obj
obj3.obj
```

To move `obj1.obj` to the end of `mylib.lib`:

```
ar51 --move mylib.lib obj1.obj
```

To move `obj3.obj` just before `obj2.obj`:

```
ar51 -m -b obj3.obj mylib.lib obj2.obj
```

The library `mylib.lib` after these two invocations now looks like:

```
obj3.obj
obj2.obj
obj1.obj
```

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.lib obj1.obj
-w5
```

Specify the option file to the archiver:

```
ar51 --option-file=myoptions
```

This is equivalent to the following command line:

```
ar51 -x mylib.lib obj1.obj -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

```
ar51 --print mylib.lib
```

The archiver prints a list of all object modules in the library `mylib.lib`:

```
ar51 -t -s0 mylib.lib
```

The archiver prints per object all symbols in the library. For example:

```
cstart.obj
   symbols:
        __cstart
        .vector.0
        _cstart_trap
```

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.lib` contains the following object (see option **--print**):

`obj1.obj`

To add `obj2.obj` to the end of `mylib.lib`:

`ar51 --replace mylib.lib obj2.obj`

To insert `obj3.obj` just before `obj2.obj`:

```
ar51 -r -b obj2.obj mylib.lib obj3.obj
```

The library `mylib.lib` after these two invocations now looks like:

```
obj1.obj
obj3.obj
obj2.obj
```

Creating a new library

To *create a new library file*, add an object file and specify a library that does not yet exist:

```
ar51 --replace newlib.lib obj1.obj
```

The archiver creates the library `newlib.lib` and adds the object `obj1.obj` 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**):

```
ar51 -r -n mylib.lib obj1.obj
```

The archiver overwrites the library `mylib.lib` and adds the object `obj1.obj` to it. The new library `mylib.lib` only contains `obj1.obj`.

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.

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:

```
ar51 --extract --warning=5 mylib.lib obj1.obj
```

Related information

-

9.9. HLL Object Dumper Options

The high level language (HLL) dumper **hldump51** 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:

```
hldump51 -FdhMsy test.elf
hldump51 --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: --blank-out (-b)

Command line syntax

--blank-out[**=***flag*]

-b[*flag*]

You can specify the following format flags:

+/-labels	**l/L**	Black out hexadecimal address and labels.

Default: **--blank-out=L**

Description

With this option you can blank out addresses and optionally labels in all dump phases. Instead of the addresses and labels crosses (X's) are shown.

The **+labels** sub-option blanks out hexadecimal addresses and labels. With the **-labels** sub-option only hexadecimal addresses are blanked out. This is the default.

This option is useful when you want to compare the output, but want to ignore the addresses and labels.

Example

```
hldump51 -F2 hello.elf

----------- Section dump ----------


                                      .rseg 'code_main' code, at(0x000004ef)
000004ef 74 37     _main:            MOV    A, #55
000004f1 f5 49                       MOV    _$varargs_printf, A
000004f3 7f 2c                       MOV    R7, #44
000004f5 02 04 f8                    LJMP   _printf
                                      ; End of section

hldump51 -F2 --blank-out hello.elf

----------- Section dump ----------


                                      .rseg 'code_main' code, at(0x000004ef)
XXXXXXXX 74 37     _main:            MOV    A, #55
XXXXXXXX f5 49                       MOV    _$varargs_printf, A
XXXXXXXX 7f 2c                       MOV    R7, #44
XXXXXXXX 02 04 f8                    LJMP   _printf
                                      ; End of section

hldump51 -F2 --blank-out=+labels hello.elf
```

```
----------- Section dump ----------

                                  .rseg 'code_main' code, at(0x000004ef)
XXXXXXXX 74 37    XXXXXXXXXX:      MOV    A, #55
XXXXXXXX f5 49                     MOV    XXXXXXXXXX, A
XXXXXXXX 7f 2c                     MOV    R7, #44
XXXXXXXX 02 04 f8                  LJMP   XXXXXXXXXX
                                  ; End of section
```

Related information

-

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

By default all sections are included.

Related information

Section 7.5.2, *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.

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:

```
hldump51 --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:

```
hldump51 --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

```
hldump51 --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

```
hldump51 -F2 hello.elf

----------- Section dump ----------

                                .rseg 'code_main' code, at(0x000004ef)
000004ef 74 37    _main:        MOV    A, #55
000004f1 f5 49                  MOV    _$varargs_printf, A
000004f3 7f 2c                  MOV    R7, #44
000004f5 02 04 f8               LJMP   _printf
                                ; End of section

hldump51 -F2 --disassembly-without-encoding hello.elf

----------- Section dump ----------

              .rseg 'code_main' code, at(0x000004ef)
_main:          MOV    A, #55
                MOV    _$varargs_printf, A
                MOV    R7, #44
                LJMP   _printf
              ; End of section
```

Related information

-

HLL object dumper option: --dump-format (-F)

Command line syntax

--dump-format[**=**_flag_,...]

-F[_flag_]...

You can specify the following format flags:

+/-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.
+/-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 **DFHMSY** (nothing)
	1	Alias for **DFhMSY** (only HLL symbols)
	2	Alias for **dFHMSY** (only section contents)
	3	Alias for **dfhmsy** (everything)

Default: **--dump-format=dFhmsy**

Description

With this option you can control which parts of the dump output you want to see. By default, all parts are dumped, except for part 6.

1. Module list

2. Section list

3. Section dump (disassembly)

4. HLL symbol table

5. Assembly level symbol table

6. Debug control flow section

You can limit the number of sections that will be dumped with the options **--sections** and **--section-types**.

Related information

Section 7.5.2, *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

```
hldump51 -F1 hello.elf

---------- HLL symbol table ----------

3:00000037      6 char          __data world[6] [hello.c]
3:0000003d      9 struct        __data _dbg_request [dbg.c]

hldump51 -e -F1 hello.elf

---------- HLL symbol table ----------

3:00000037      6 char          __data world[6] [hello.c]
```

```
3:0000003d     9 struct        __data _dbg_request [dbg.c]
3:0000003d     2  int                   _errno
3:0000003f     1  enum                  nr
3:00000040     6  union                 u
3:00000040     2    struct              exit
3:00000040     2     int                status
3:00000040     3    struct              open
3:00000040     1     const char  __data * pathname
3:00000041     2     unsigned short int   flags
   ...

hldump51 -eb -F1 hello.elf

---------- HLL symbol table ----------

3:00000037     6 char          __data world[6] [hello.c]
3:00000037     1  char
3:00000038     1  char
3:00000039     1  char
3:0000003a     1  char
3:0000003b     1  char
3:0000003c     1  char
3:0000003d     9 struct        __data _dbg_request [dbg.c]
3:0000003d     2  int                   _errno
3:0000003f     1  enum                  nr
3:00000040     6  union                 u
3:00000040     2    struct              exit
3:00000040     2     int                status
3:00000040     3    struct              open
3:00000040     1     const char  __data * pathname
3:00000041     2     unsigned short int   flags
   ...

hldump51 -ef -F1 hello.elf

---------- HLL symbol table ----------

3:00000037     6 char          __data world[6] [hello.c]
3:0000003d     9 struct        __data _dbg_request [dbg.c]
3:0000003d     2  int                   _dbg_request._errno
3:0000003f     1  enum                  _dbg_request.nr
3:00000040     6  union                 _dbg_request.u
3:00000040     2    struct              _dbg_request.u.exit
3:00000040     2     int                _dbg_request.u.exit.status
3:00000040     3    struct              _dbg_request.u.open
3:00000040     1     const char  __data * _dbg_request.u.open.pathname
3:00000041     2     unsigned short int   _dbg_request.u.open.flags
   ...
```

```
hldump51 -eg -F1 hello.elf

---------- HLL symbol table ----------

3:00000037      6 char          __data world[6] [hello.c]
3:0000003d      9 struct        __data _dbg_request [dbg.c]
3:0000003d      2   int                    _errno
3:0000003f      1   enum                   nr
3:00000040      6   union                  u
3:00000040      2     struct                 exit
3:00000040      2       int                      status
3:00000042      4                              {gap}
3:00000040      3     struct                 open
3:00000040      1       const char  __data * pathname
3:00000041      2       unsigned short int   flags
3:00000043      3                               {gap}
   ...

hldump51 -en -F1 hello.elf

---------- HLL symbol table ----------

3:00000037      6 char          __data world[6] [hello.c]
3:0000003d      9 struct        __data _dbg_request [dbg.c]
3:0000003d      2 | int                    _errno
3:0000003f      1 | enum                   nr
3:00000040      6 | union                  u
3:00000040      2 | | struct                 exit
3:00000040      2 | | | int                      status
3:00000040      3 | | struct                 open
3:00000040      1 | | | const char  __data * pathname
3:00000041      2 | | | unsigned short int   flags
   ...
```

Related information

Section 7.5.2, *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:

```
hldump51 -?
hldump51 --help
hldump51
```

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 `.db`, `.dw`, `.dl` kind of directives, depending on the size of the data. RAM sections are represented with `.ds` 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

```
hldump51 -F2 --section=[data__1_str] hello.elf

---------- Section dump ----------

    .rseg '[data__1_str]' data, at (0x0537)
    .db 48,65,6c,6c,6f,20,25,73,21,0a,00                ; Hello %s!..
  ; End of section

hldump51 -F2 --section=[data__1_str] --hex hello.elf

---------- Section dump ----------

                        section 57 ([data__1_str]):
00000537 48 65 6c 6c 6f 20 25 73 21 0a 00               Hello %s!..
```

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:

```
hldump51 --option-file=myoptions
```

This is equivalent to the following command line:

```
hldump51 --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

hldump51 --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: --sections (-s)

Command line syntax

--sections=_name_**,...**

-s_name_**,...**

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 7.5.2, *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:

```
hldump51 --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

```
hldump51 -F2 hello.elf

----------- Section dump ----------

                                    .rseg '$interrupt_0' code, at(0x00000000)
00000000 02 04 4f   _$interrupt_0:   LJMP     __start
                                    ; End of section

hldump51 --symbols=none -F2 hello.elf

----------- Section dump ----------

                                    .rseg '$interrupt_0' code, at(0x00000000)
00000000 02 04 4f                    LJMP     code:0x44f
                                    ; End of section
```

Related information

Section 7.5.2, *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

`hldump51 --output-type=xml --output=hello.xml ../hello.elf`

The field `<File name="../hello.elf">` is used in `hello.xml`.

`hldump51 --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**

9.10. Expire Cache Utility Options

With the utility **expire51** 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 10.4, *Compiler Cache*.

The invocation syntax is:

expire51 [*option*]... *cache-directory*

The compiler cache is present in the directory c51cache 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

expire51 --access --days=7 "*installation-dir*\mproject\.cache"

Related information

-

Expire cache utility option: --days (-d)

Menu entry

1. Select **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

-dn

Description

Remove all files older than n days from the cache.

Example

To remove all files older than seven days, enter:

```
expire51 --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:

```
expire51 --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 `expire51_errors.html`, use redirection and enter:

```
expire51 --diag=html:all > expire51_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:

```
expire51 -?
expire51 --help
expire51
```

To see a detailed description of the available options, enter:

```
expire51 --help=options
```

Related information

-

Expire cache utility option: --megabytes (-m)

Menu entry

1. Select **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

-mm

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:

```
expire51 --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:

```
expire51 --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

```
expire51 -t "installation-dir\mproject\.cache"

installation-dir\mproject\.cache\c51cache:
1 MB, 2 directories, 2 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

```
expire51 -v --megabytes=0 "installation-dir\mproject\.cache"

2014-07-03 12:36:17 installation-dir\mproject\.cache\c51cache\myproject\6f0a3ba4
```

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

-

Chapter 10. 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.

10.1. 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.

10.2. 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.

10.3. 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.

10.4. 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 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 `c51cache` 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 **expire51** 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 7.6, *Expire Cache Utility*.

10.5. 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

10.6. 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 Behavior tab, select **Enable 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.

10.7. Number of Sections

The linker speed depends on the number of sections in the object files. The more sections, the longer the locating will take. You can decrease the link time by creating output sections in the LSL file. For example:

```
section_layout ::code
{
 group (ordered)
 {
  section "code_output1" ( size = 64k, attributes = x, fill=0xFF,
                             overflow = "code_output2")
  {
   select "*__cocofun*";
  }
 }
}
```

Chapter 11. 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 11.1, *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 11.2, *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 8051 toolset for each core variant. Both Eclipse and the control program **cc51** automatically select the appropriate libraries depending on the specified options.

Libraries	Description
c51*m*{r | s}[*b*].lib	C libraries for each model *m*: s (small), a (aux), l (large) Optional letters: r = reentrant s = static *b* = bank number 0, 1, 2, 3
fp51*m*{r | s}[t].lib	Floating-point libraries for each model *m*: s (small), a (aux), l (large) (contains floating-point functions needed by the C compiler) Optional letters: r = reentrant s = static t = trapping (control program option **--fp-model=+trap**)
rt51.lib	Run-time library (contains other run-time functions needed by the C compiler)

Sources for the libraries are present in the directory `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 (`fp51m{r | s}t.lib`), 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.

11.1. 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\c51 –DWCHAR_SUPPORT_ENABLED
```

11.1.1. assert.h

`assert(`*expr*`)`   Prints a diagnostic message if NDEBUG is not defined. (Implemented as macro)

11.1.2. 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]).
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.

ctype.h	wctype.h	Description
_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.

11.1.3. 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

11.1.4. 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

11.1.5. 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)*

11.1.6. fenv.h

Contains mechanisms to control the floating-point environment. The functions in this header file are not implemented.

`fegetenv`	Stores the current floating-point environment. (*Not implemented*)
`feholdexept`	Saves the current floating-point environment and installs an environment that ignores all floating-point exceptions. (*Not implemented*)
`fesetenv`	Restores a previously saved (fegetenv or feholdexcept) floating-point environment. (*Not implemented*)
`feupdateenv`	Saves the currently raised floating-point exceptions, restores a previously saved floating-point environment and finally raises the saved exceptions. (*Not implemented*)
`feclearexcept`	Clears the current exception status flags corresponding to the flags specified in the argument. (*Not implemented*)
`fegetexceptflag`	Stores the current setting of the floating-point status flags. (*Not implemented*)

`feraiseexcept`	Raises the exceptions represented in the argument. As a result, other exceptions may be raised as well. (*Not implemented*)
`fesetexceptflag`	Sets the current floating-point status flags. (*Not implemented*)
`fetestexcept`	Returns the bitwise-OR of the exception macros corresponding to the exception flags which are currently set *and* are specified in the argument. (*Not implemented*)

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. (*Not implemented*)
`fesetround`	Sets the current rounding directions. (*Not implemented*)

Currently no rounding mode macros are implemented.

11.1.7. float.h

The header file `float.h` defines the characteristics of the real floating-point types `float`, `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 11.1.14, *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 11.1.14, *math.h and tgmath.h*)

11.1.8. 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`)

11.1.9. 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*)

11.1.10. 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   ^=
```

11.1.11. limits.h

Contains the sizes of integral types, defined as macros.

11.1.12. 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.

11.1.13. 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 11.1.22, *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.

11.1.14. 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. (*Not implemented*)
expm1	expm1f	expm1l	expm1	Returns the result of the exponential function e^x-1. (*Not implemented*)
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`. (*Not implemented*)
log2	log2f	log2l	log2	Returns the base-2 logarithm of `x.x>0`. (*Not implemented*)
ilogb	ilogbf	ilogbl	ilogb	Returns the signed exponent of x as an integer. `x>0`. (*Not implemented*)
logb	logbf	logbl	logb	Returns the exponent of `x` as a signed integer in value in floating-point notation. `x > 0`. (*Not implemented*)

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 $0.5 \leq \|f\| \leq 1.0$ 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: $\|f\| < 1.0$ and *f*+*n*=`x`. Returns *f*, stores *n*.
scalbn	scalbnf	scalbnl	scalbn	Computes the result of $x*FLT\_RADIX^n$. 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`. (*Not implemented*)

math.h			tgmath.h	Description
lrint	lrintf	lrintl	lrint	Returns the rounded integer value as a `long int` according to the current rounding direction. See `fenv.h`. (*Not implemented*)
llrint	llrintf	llrintl	llrint	Returns the rounded integer value as a `long long int` according to the current rounding direction. See `fenv.h`. (*Not implemented*)
nearbyint	nearbyintf	nearbyintl	nearbyint	Returns the rounded integer value as a floating-point according to the current rounding direction. See `fenv.h`. (*Not implemented*)
round	roundf	roundl	round	Returns the nearest integer value of x as int. (*Not implemented*)
lround	lroundf	lroundl	lround	Returns the nearest integer value of x as long int. (*Not implemented*)
llround	lroundf	llroundl	llround	Returns the nearest integer value of x as long long int. (*Not implemented*)
trunc	truncf	truncl	trunc	Returns the truncated integer value x. (*Not implemented*)

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.
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. (*Not implemented*)
remquo	remquof	remquol	remquo	Same as remainder. In addition, the argument *quo is given a specific value (see ISO). (*Not implemented*)

Power and absolute-value functions

math.h			tgmath.h	Description
cbrt	cbrtf	cbrtl	cbrt	Returns the real cube root of x $(=x^{1/3})$. (*Not implemented*)
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. (*Not implemented*)
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).
sqrt	sqrtf	sqrtl	sqrt	Returns the non-negative square root of x. x 0.

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*. (*Not implemented*)
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. (*Not implemented*)
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. (*Not implemented*)

Positive difference, maximum, minimum

math.h			tgmath.h	Description
fdim	fdimf	fdiml	fdim	Returns the positive difference between: $\lvert x-y \rvert$. (*Not implemented*)
fmax	fmaxf	fmaxl	fmax	Returns the maximum value of their arguments. (*Not implemented*)
fmin	fminf	fminl	fmin	Returns the minimum value of their arguments. (*Not implemented*)

Error and gamma (Not implemented)

math.h			tgmath.h	Description
erf	erff	erfl	erf	Computes the error function of x. (*Not implemented*)
erfc	erfcf	erfcl	erc	Computes the complementary error function of x. (*Not implemented*)
lgamma	lgammaf	lgammal	lgamma	Computes the $*\log_e \lvert \Gamma(x) \rvert$ (*Not implemented*)
tgamma	tgammaf	tgammal	tgamma	Computes $\Gamma(x)$ (*Not implemented*)

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.

11.1.15. 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()`.

11.1.16. 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.

11.1.17. 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.

11.1.18. 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
```

11.1.19. 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 0 (zero).

offsetof(_type,   Expands to an integer constant expression with type size_t that is the offset
_member)   in bytes of _member within structure type _type.

11.1.20. stdint.h

See Section 11.1.8, *inttypes.h and stdint.h*

11.1.21. 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 0 (zero).
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*)

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*)

stdio.h	Description
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
%	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
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.
[...]	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*)

stdio.h	wchar.h	Description
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 11.1.17, *stdarg.h*)
vscanf(*format*, *arg*)	vwscanf(*format*, *arg*)	Same as sscanf/swscanf, but extra arguments are given as variable argument list *arg*. (See Section 11.1.17, *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 11.1.17, *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)*
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 11.1.17, *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 11.1.17, *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 11.1.17, *stdarg.h*)

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)*

stdio.h	wchar.h	Description
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*)
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.

stdio.h	Description
`perror(*s)`	Prints *s* and the error message belonging to the integer `errno`. (See Section 11.1.4, *errno.h*)

11.1.22. 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 11.1.12, *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
        (*s,**endp,base)
	long long wcstoll
	            (*s,**endp,base)

unsigned long strtoul
        (*s,**endp,base)
	unsigned long wcstoul
	            (*s,**endp,base)

unsigned long long strtoull
        (*s,**endp,base)
	unsigned long long wcstoull
	            (*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 11.1.16, *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 11.1.26, *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 11.1.26, *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 11.1.26, *wchar.h*).

11.1.23. 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*.
`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 11.1.12, *locale.h*)

string.h	wchar.h	Description
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 11.1.4, *errno.h*)
strlen(*s)	wcslen(*s)	Returns the length of string *s*.

11.1.24. 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 11.1.12, *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

%%   %

11.1.25. 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*)

11.1.26. wchar.h

Many functions in `wchar.h` represent the wide-character variant of other functions so these are discussed together. (See Section 11.1.21, *stdio.h and wchar.h*, Section 11.1.22, *stdlib.h and wchar.h*, Section 11.1.23, *string.h and wchar.h* and Section 11.1.24, *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 11.1.22, *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 11.1.22, *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 11.1.22, *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*.

11.1.27. wctype.h

Most functions in `wctype.h` represent the wide-character variant of functions declared in `ctype.h` and are discussed in Section 11.1.2, *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 11.1.12, *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 11.1.12, *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")`
`towupper(wc)`	`towctrans(wc,wctrans("toupper")`

11.2. 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`	-
`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`	-

Function	Not reentrant because
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	(*Not implemented*)
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.
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.

Function	Not reentrant because
`ctanh ctanhf ctanhl`	Sets errno via calls to other functions.
`ctime`	Calls asctime
`difftime`	-
`div ldiv lldiv`	-
`erf erfl erff`	(*Not implemented*)
`erfc erfcf erfcl`	(*Not implemented*)
`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`	(*Not implemented*)
`expm1 expm1f expm1l`	(*Not implemented*)
`fabs fabsf fabsl`	-
`fclose`	Uses values in iob[ ]. See (1).
`fdim fdimf fdiml`	(*Not implemented*)
`feclearexcept`	(*Not implemented*)
`fegetenv`	(*Not implemented*)
`fegetexceptflag`	(*Not implemented*)
`fegetround`	(*Not implemented*)
`feholdexept`	(*Not implemented*)
`feof`	Uses values in iob[ ]. See (1).
`feraiseexcept`	(*Not implemented*)
`ferror`	Uses values in iob[ ]. See (1).
`fesetenv`	(*Not implemented*)
`fesetexceptflag`	(*Not implemented*)
`fesetround`	(*Not implemented*)
`fetestexcept`	(*Not implemented*)
`feupdateenv`	(*Not implemented*)
`fflush`	Modifies iob[ ]. See (1).
`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`	(*Not implemented*)
`fmax fmaxf fmaxl`	(*Not implemented*)
`fmin fminf fminl`	(*Not implemented*)

Function	Not reentrant because
`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`	(*Not implemented*)
`imaxabs`	-
`imaxdiv`	-
`isalnum iswalnum`	-
`isalpha iswalpha`	-
`isascii iswascii`	-
`iscntrl iswcntrl`	-
`isdigit iswdigit`	-
`isfinite`	-
`isgraph iswgraph`	-
`isgreater`	-
`isgreaterequal`	-

Function	Not reentrant because
`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`	(*Not implemented*)
`llrint llrintf llrintl`	(*Not implemented*)
`llround llroundf llroundl`	(*Not implemented*)
`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`	(*Not implemented*)
`log2 log2f log2l`	(*Not implemented*)

Function	Not reentrant because
`logb logbf logbl`	(*Not implemented*)
`longjmp`	-
`lrint lrintf lrintl`	(*Not implemented*)
`lround lroundf lroundl`	(*Not implemented*)
`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`	(*Not implemented*)
`nearbyint nearbyintf nearbyintl`	(*Not implemented*)
`nextafter nextafterf nextafterl`	(*Not implemented*)
`nexttoward nexttowardf nexttowardl`	(*Not implemented*)
`offsetof`	-
`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)

Function	Not reentrant because
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	(*Not implemented*)
remove	Uses global File System Simulation buffer, _dbg_request
remquo remquof remquol	(*Not implemented*)
rename	Uses global File System Simulation buffer, _dbg_request
rewind	Eventually calls _lseek
rint rintf rintl	(*Not implemented*)
round roundf roundl	(*Not implemented*)
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
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	-

Function	Not reentrant because
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	(*Not implemented*)
time	Uses static variable which defines initial start time
tmpfile	Uses iob[ ]. See (1).
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	-

Function	Not reentrant because
towupper	-
trunc truncf truncl	(*Not implemented*)
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.

Chapter 12. List File Formats

This chapter describes the format of the assembler list file and the linker map file.

12.1. 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 5.5, *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 8051: Assembler vx.yrz Build yymmddqq
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 $[NO]LIST, $PAGELENGTH, $PAGEWIDTH, $[NO]PAGING, 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
                          .
                          .
0000                      9 _main:
                         10          .using 0
0000 7Frr    1    1      12          mov     R7,#LOW(_$1$str)
0002 7Err    1    2      13          mov     R6,#HIGH(_$1$str)
0004 12rrrr  2    4      14          gcall   _printf
                          .
                          .
0000                     38          .ds     2
   |   RESERVED
0001
```

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.

12.2. 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 (`.obj`) to output sections. Locate information is not present, because that is not available for an 8051 project. 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 6.9, *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.
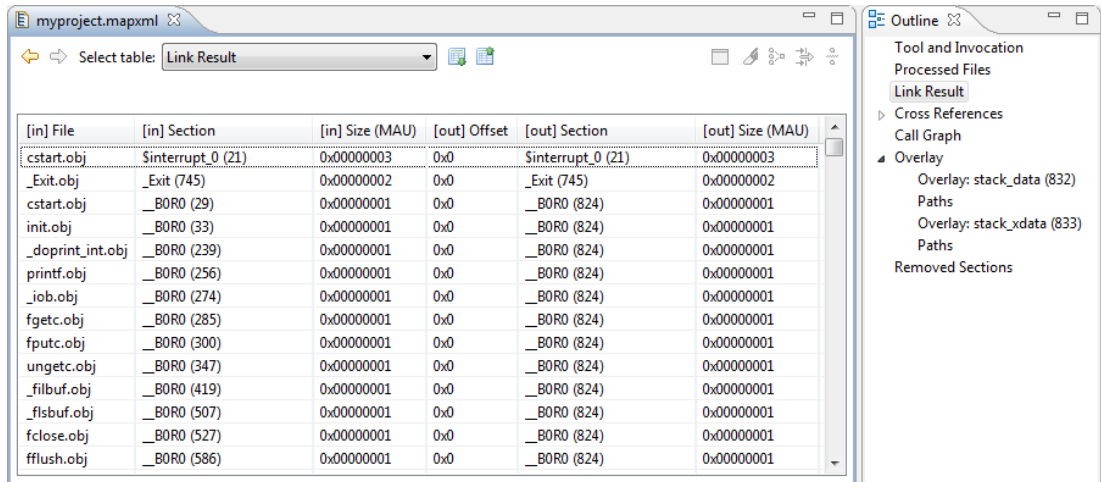
Each page displays a part of the map file. You can use the drop-down list or the Outline view to navigate through the different tables and you can use the following buttons.

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.

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 (.obj) 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 [s0:2,24; s1:0,0]
|
+-- _main [s0:2,22; s1:0,0]
|    |
|    +-- _printf [s0:2,20; s1:0,0]
|        |
|        +-- __doprint [s0:2,18; s1:0,0]
|            |
|            +-- _doprint_noarg.src:_$cocofun_1 [s0:2,2; s1:0,0]
|            |
|            +-- _doprint_noarg.src:_$cocofun_2 [s0:2,2; s1:0,0]
|            |
|            +-- __io_putc [s0:2,16; s1:0,0]
|                |
|                +-- _fputc [s0:2,14; s1:0,0]
|                |   |
|                |   +-- __flsbuf [s0:2,12; s1:0,0]
|                |   |   |
|                |   |   +-- _flsbuf.src:_$cocofun_12 *
```

```
|               |   |   |
|               |   |   +-- _flsbuf.src:_$cocofun_8 [s0:2,4; s1:0,0]
|               |   |   |   |
|               |   |   |   +-- _flsbuf.src:_$cocofun_18 *
|               |   |   |   |
|               |   |   |   +-- __INCIXD *
|               |   |   |
```

- A * after a function name indicates that the call tree starting with this function is shown separately, with a * in front of the function name.

- A * in front of a function name indicates that the function is not considered a "root" in the call graph since it is called by one or more other functions.

- 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.

- An '__INDIRECT__' entry (not shown in this example) indicates an indirect function call. It is not an actual function. Each function listed as a caller of the __INDIRECT__ placeholder symbol places a call through a function pointer. Each function listed as a callee of the __INDIRECT__ placeholder symbol has its address taken (and used).

- [ ] 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 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.
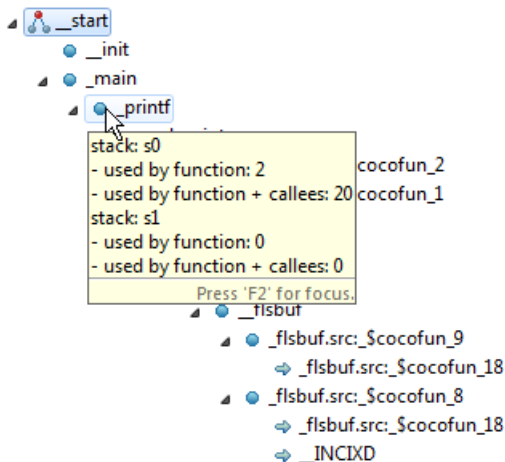
Icon	Meaning	Description
🕸	Root	This function is the top of the call graph. If there are interrupt handlers, there can be several roots.

Icon	Meaning	Description
🐾	Callee	This function is referenced by several No leaf 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.
●	Node	A normal node (function) in the call graph.
➡	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.

Overlay

This part of the map file shows how the static stack is organized. This part also shows the locate overlay information if you used overlay groups in the linker script file.

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 expand or collapse a part of the information.

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 13. Object File Formats

This chapter describes the format of several object files.

13.1. ELF/DWARF Object Format

The TASKING VX-toolset for 8051 by default produces objects in the ELF/DWARF 3 format.

For a complete description of the ELF format, please refer to the *Tool Interface Standard (TIS)*.

For a complete description of the DWARF format, please refer to the *DWARF Debugging Information Format Version 3*. See http://dwarfstd.org/

The implementation of the ELF object format and the DWARF 3 debug information for the TASKING VX-toolset for 8051 is described in the TASKING 8051 ELF/DWARF Application Binary Interface (EDABI) v1.2 [2018, Altium].

13.2. 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 see linker option **--output**.

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:

```
:020000040000FA
 | |   | |   |_ 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:

```
:0400000500001604DD
 | |   | |         |_ 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
```

13.3. Motorola S-Record Format

To generate a Motorola S-record output file see linker option **--output**.

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 k 5 1
S00700006C6B3531BB
```

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:

```
S3070000FFFFE6E6825
   | |         |   |_ 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:

```
S70500001604E0
    | |         |_checksum
    | |_ address
    |_ length
```

Chapter 14. 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.

14.1. 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.

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 14.4, *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 14.5, *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 14.6, *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 14.6.3, *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 14.8, *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
}
```

14.2. 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.

14.2.1. Preprocessing

When the linker loads an LSL file, the linker first processes it with a C-style prepocessor. As such, it strips C and C++ comments. Lines starting with the `#` character are taken as commands for the preprocessor. You can use the standard ISO C99 preprocessor directives, including:

```
#include "file"
#include <file>
```

Preprocess and include file *file* at this point in the LSL file.

For example:

```
#include "arch.lsl"
```

Preprocess and include the file `arch.lsl` at this point in the LSL file.

```
#if condition
#else
#endif
```

If the *condition* evaluates to a non-zero value, copy the following lines, up to an `#else` or `#endif` command, skip lines between `#else` and `#endif`, if present. If the condition evaluates to zero, skip the lines up to the `#else` command, or `#endif` if no `#else` is present, and copy the lines between the `#else` and `#endif` commands.

```
#ifdef identifier
#else
#endif
```

Same as `#if`, but with `defined(identifier)` as condition.

```
#error text
```

Causes a fatal error the given message (optional).

14.2.2. 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 '//'.

14.2.3. 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.

14.2.4. 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
```

14.2.5. 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
                     | mem : mem_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 offset of *addr_id*, which is a named section, group, or memory in the address space of the section layout. If the referenced object is a group or memory, it must be defined in the LSL file. 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,
    addressof( mem:foo ),
    addressof( mem:foo ) + sizeof( mem:foo ) )
```

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. If the referenced object is a group or memory, it must be defined in the LSL file.

14.2.6. 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.

14.2.7. 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.

14.2.8. 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
                      | *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
                       |  attributes
                       |  tag
```

- The **min_size** statement must be present.

- The **min_size** value must be 1 or greater.

- 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
                     | tag
```

- 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.

```
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
```

14.2.9. 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.

14.2.10. 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.

14.2.11. Section Setup

```
section_setup       ::= section_setup space_ref <tag_attr>^{0|1}
                        { <section_setup_item>^{>=0} }

section_setup_item
                    ::= reserved_range
                      | stack_def ;
                      | heap_def ;
                      | copy_table_def ;
                      | start_address ;
```

```
                          |  reference_space_restriction ;
                          |  modify linktime_modification

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.

14.2.12. 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
```
$$::= \textbf{select} \texttt{ <ref\_tree>}^{0|1} \texttt{ <section\_name>}^{0|1}$$
$$\texttt{<section\_selections>}^{0|1}$$

- Either a *section_name* or at least one *section_selection* must be defined.

```
section_selections
```
$$::= \texttt{(} \textit{section\_selection}$$
$$\texttt{<, section\_selection>}^{>=0} \texttt{)}$$

```
section_selection
```
$$::= \textbf{attributes =} \texttt{ < <+|-> } \textit{attribute>}^{>0}$$
$$| \textit{tag}$$

- **+***attribute* means: select all sections that have this attribute.

- **-***attribute* means: select all sections that do not have this attribute.

```
special_section_statement
```
$$::= \textbf{heap} \textit{ heap\_name } \texttt{<stack\_heap\_mods>}^{0|1}$$
$$| \textbf{stack} \textit{ stack\_name } \texttt{<stack\_heap\_mods>}^{0|1}$$
$$| \textbf{copytable}$$
$$| \textbf{reserved} \textit{ section\_name } \texttt{<reserved\_specs>}^{0|1}$$

- Special sections cannot be selected in load-time groups.

$$\textit{stack\_heap\_mods} ::= \texttt{(} \textit{ stack\_heap\_mod } \texttt{<, stack\_heap\_mod>}^{>=0} \texttt{)}$$

```
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
                            | tag
```

- The **allow-cross-references** property is only allowed for *overlay* groups.

- The **copy** and **nocopy** properties cannot be applied both to the same group.

- 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.

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**
                        | **nocopy**
                        | tag

$struct\_statement$
                    ::= **struct {** <$struct\_item$>$^{>=0}$ **}**

$struct\_item$ ::= $expr$ **:** $number$ **;**

14.3. 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.

14.4. 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
      attributes    b
   heap
      min_size
      grows          low_to_high   high_to_low
      align
      fixed
      id
      attributes    b
   copytable
      align
      copy_unit
      dest
      page
   reserved
   start_address
      run_addr
```

```
      symbol
   map

   map
      dest           bus   space
      dest_dbits
      dest_offset
      size
      src_dbits
      src_offset
      priority
      exec_priority
```

14.4.1. 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
}
```

14.4.2. 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 and must be non-zero.

- 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 14.4.4, *Mappings*.

```
bus bus_name
{
    mau = 8;
    width = 8;
    map ( map_description );
}
```

14.4.3. 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 and must be non-zero.

- 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 14.8.2, *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 14.4.4, *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 14.8.3, *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.

  A stack may have an **attributes** property with value **b**. Such a stack must be cleared at program startup. No other attributes are allowed.

  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 14.8.3, *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 14.8, *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.

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 14.8.3, *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 );
}
```

14.4.4. 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).

A mapping can optionally have a name which can be referenced in an address assignment.

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.

14.5. 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
```

14.5.1. 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
}
```

14.5.2. 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
}
```

14.5.3. 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 14.6.3, *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 14.8.2, *Creating and Locating Groups of Sections*).

- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the memory. This field is required and must be non-zero.

- 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 14.4.4, *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 14.4.2, *Defining Internal Buses*.

14.6. 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
```

14.6.1. 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
}
```

14.6.2. 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);
}
```

14.6.3. 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 14.5.3, *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 14.4.2, *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.

14.7. 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
   reserved
   start_address
      run_addr
      symbol
   prohibit_references_to
   modify input
      space
      attributes
      copy
```

14.7.1. Setting up a Section

With the keyword **section_setup** you can define stacks, heaps, copy tables, start address and/or reserved address ranges outside their address space definition. In addition you can configure space reference restrictions and input section modifications.

```
section_setup ::my_space
{
   reserved address range
   stack definition
   heap definition
   copy table definition
   start adress
   space reference restrictions
   input section modifications
}
```

See the subsections Stacks and heaps, Copy tables, Start address and Reserved address ranges in Section 14.4.3, *Defining Address Spaces* for details on the keywords **stack**, **heap**, **copytable** 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.

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

- add (+p) or remove (-p) the protected attribute

You cannot set the protected attribute on linker created sections like reserved sections and output sections.

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.

14.8. 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
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
```

14.8.1. 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
}
```

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.

14.8.2. 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 14.8.3, *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.

When you use wildcards, the linker has the possibility to skip some sections in the selection process when it is obvious you did not want to select these. For example, inlined vector sections, but also a start section already having an absolute start address.

For example, when you specify restrictions on code sections excluding vector handler code, you should use a wildcard to select the code sections. select "code*"; will not select vector handler code sections, whereas select "code"; does.

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.

- 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 (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. You cannot apply both `copy` and `nocopy` to the same statement.

2. Define the mutual order of sections in an LSL 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.

    > Note that when you use the linker optimization option **--optimize=+copytable-compression**, unrestricted sections affected by the copy table are located as if they were in a clustered LSL group. This option is enabled by default.

    - 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 (ordered, run_addr = 0xa00f0000)
```

A group with an absolute address must be 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 (ordered, run_addr = mem:A[0x1000])
```

A group with an absolute address must be 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 14.4.3, *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";
}
```

14.8.3. 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 14.4.3, *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**, **nocopy** 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;
  addressof( mem:foo )                        : 4;
  addressof( mem:foo ) + sizeof( mem:foo )    : 4;
  checksum( crc32w,
    addressof( mem:foo ),
    addressof( mem:foo ) + sizeof( mem:foo ) ) : 4}
}
```

14.8.4. 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
}
```

14.8.5. 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 15. 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.

15.1. 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.

15.2. 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:

```
void my_function(int a);
int main(void)
{
  int (*new_function)(int a) = my_function;
  return (*new_function)(10); /* the behavior is undefined */
}
```

15.3. 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.

15.4. 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.

15.5. 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.

15.6. 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.

15.7. 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.

15.8. 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.

15.9. 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.

15.10. 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.

15.11. 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 16. MISRA C Rules

This chapter contains an overview of the supported and unsupported MISRA C rules.

16.1. MISRA C:1998

This section lists all supported and unsupported MISRA C:1998 rules.

See also Section 3.7.2, *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.

16.2. MISRA C:2004

This section lists all supported and unsupported MISRA C:2004 rules.

See also Section 3.7.2, *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.

16.3. MISRA C:2012

This section lists all supported and unsupported MISRA C:2012 rules.

See also Section 3.7.2, *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.

12.5 (M) The `sizeof` operator shall not have an operand which is a function parameter declared as "array of type".

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) The 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` 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.

21.13 (M)   Any value passed to a function in `<ctype.h>` shall be representable as an `unsigned char` or be the value `EOF`.

21.14 (R)   The Standard Library function `memcmp` shall not be used to compare null terminated strings.

21.15 (R)   The pointer arguments to the Standard Library functions `memcpy`, `memmove` and `memcmp` shall be pointers to qualified or unqualified versions of compatible types

21.16 (R)   The pointer arguments to the Standard Library function `memcmp` shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type

21.17 (M)   Use of the string handling functions from `<string.h>` shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters.

21.18 (M)   The `size_t` argument passed to any function in `<string.h>` shall have an appropriate value.

21.19 (M)   The pointers returned by the Standard Library functions `localeconv`, `getenv`, `setlocale` or, `strerror` shall only be used as if they have pointer to const-qualified type.

21.20 (M)   The pointer returned by the Standard Library functions `asctime`, `ctime`, `gmtime`, `localtime`, `localeconv`, `getenv`, `setlocale` or `strerror` shall not be used following a subsequent call to the same function.

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.

x   22.7   (R)   The macro `EOF` shall only be compared with the unmodified return value from any Standard Library function capable of returning `EOF`.

x   22.8   (R)   The value of `errno` shall be set to zero prior to a call to an errno-setting-function.

x   22.9   (R)   The value of `errno` shall be tested against zero after calling an errno-setting-function.

x   22.10   (R)   The value of `errno` shall only be tested when the last function to be called was an errno-setting-function.