TASKING C166
ELF-DWARF APPLICATION BINARY INTERFACE

Document ID:	119-EDABI
Status:	Released
Version:	1.4
Date:	2008-09-04

A L T I U M B V

Contents

Revision History

- v1.0: Initial version
- v1.1: Version made available with the last v1.0 beta of the TASKING VX-toolset for C166. Switched formally to DWARF 3.0
- v1.2: First used in TASKING VX-toolset for C166 v2.1r1. Added SHF_TASKING_PROTECTED. Changed values of EF_C166_DATA_*. Added return_address_register in DWARF information. Updated call stack frame section.
- v1.3: First used in TASKING VX-toolset for C166 v2.1r2. Removed RLn, RHn and Rn.m from the DWARF register mapping. Updated call stack frame section. Many changes in the DWARF Call Frame Information
- v1.4: First used in TASKING VX-toolset for C166 v2.3r1. Added new DW_AT_address_class attribute table. Documented use of DW_TAG_packed_type. Added section about code compaction.

Introduction

This document describes the implementation of the ELF object format and the DWARF 3 debug information for the TASKING VX-toolset for C166. The implementation is based on:

- System V Application Binary Interface - DRAFT - 17 December 2003
  see http://www.caldera.com/developers/gabi/2003-12-17/contents.html
- DWARF Debugging Information Format, Version 3, December 20, 2005
  see http://dwarf.freestandards.org

1 ELF Implementation

1.1 ELF Header

The following paragraphs define C166 specific items in the ELF header.

1.1.1 e_ident field

The e_ident field values are defined as follows:

Field	Value	Description
e_ident[EI_CLASS]	ELFCLASS32	Identifies 32 bit architecture.
e_ident[EI_DATA]	ELFDATA2LSB	Identifies 2's complement little endian data encoding.

1.1.2 E_MACHINE

The E_MACHINE is defined as follows:

E_MACHINE	Value	Description
EM_C166	116	Infineon C16x/XC16x processor

1.1.3 E_FLAGS

The E_FLAGS field will be used to distinguish between memory models and extended architectures:

Bit	Type	Values	Meaning
0-3	EF_C166_CORE_UNDEFINED	0	Architecture not defined
	EF_C166_CORE_8X166	1	Classic 8xC166
	EF_C166_CORE_C16X	2	Infineon C16x
	EF_C166_CORE_ST10	3	STMicroelectronics ST10
	EF_C166_CORE_ST10MAC	4	STMicroelectronics ST10 with MAC unit (e.g., ST10x272)
	EF_C166_CORE_XC16X	5	Infineon XC16X
	EF_C166_CORE_SUPER10	6	STMicroelectronics Super10
	EF_C166_CORE_SUPER10M345	7	STMicroelectronics Super10M345 and derivatives
	EF_C166_CORE_C166SV1	8	Infineon C166S V1 core
		9-15	reserved for future use
4-7	EF_C166_DATA_UNDEFINED	0	Data model not defined
	EF_C166_DATA_NEAR	1	Near data model
	EF_C166_DATA_FAR	2	Far data model
	EF_C166_DATA_SHUGE	3	Segmented huge data model
	EF_C166_DATA_HUGE	4	Huge data model
		5-15	reserved for future use
8-10	EF_C166_CODE_UNDEFINED	0	Code model not defined
	EF_C166_CODE_HUGE	1	Code model with huge functions
	EF_C166_CODE_NEAR	2	Code model with near functions
		3-7	reserved for future use
11	EF_C166_SYSTEM_STACK	0	System stack is used as default for return values
	EF_C166_USER_STACK	1	User stack is used as default for return values
12	EF_C166_FLOAT_DOUBLE	0	Double precission floating point is treated as double precission
	EF_C166_FLOAT_NODOUBLE	1	Double precission floating point is treated as single precission
13-31		0	Reserved for future use

1.2 ELF Section Attribute Flags

Section attribute flags are defined in the sh_flags field of the section header record. The TASKING defined flags are in the SHF_MASKOS or the SFR_MASKPROC range:

Name	Value
SHF_MASKOS	0x0FF00000
SHF_MASKPROC	0xF0000000
SHF_TASKING_PROTECTED	0x08000000
SHF_TASKING_ABSOLUTE	0x10000000
SHF_TASKING_SEPARATE	0x20000000
SHF_TASKING_NOCLEAR	0x40000000
SHF_TASKING_PAGED	0x80000000

SHF_TASKING_PROTECTED
Sections with this flag set are protected. Sections with the SHF_TASKING_PROTECTED flag set are excluded from unreferenced section removal and duplicate section removal.

SHF_TASKING_ABSOLUTE
Sections with this flag set are absolute. The sh_addr field in the section header contains the absolute address.

SHF_TASKING_SEPARATE
Sections with the same type, attributes and name are concatenated by the linker. Sections with the SHF_TASKING_SEPARATE flag set will not be concatenated.

SHF_TASKING_NOCLEAR
These sections must have type SHT_NOBITS. Normally, sections of this type must be cleared on startup, but sections with the flag SHF_TASKING_NOCLEAR set should not be cleared.

SHF_TASKING_PAGED
Sections with this flag set are relocatable, the sh_addr field in the section header is interpreted as a page size by the linker. The section must be located within a page of this size. Pages start at a multiple of the page size. If the section name is of the form "name@group", the linker must place all sections with the same group postfix in the same page. The size of the page depends on the section type and address space.

'Max sections'
When the SHF_MERGE flag is set in combination with the SHF_TASKING_NOCLEAR flag, all sections with the same name type and flags are combined into a single section, with size equal to the largest input section. This are so-called 'max sections'.
Note that this only applies to scratch sections.

1.3 Address Spaces

Address space information for sections and symbols that is to be used by the linker is encoded in
an additional field that is added to the ELF section headers and symbol table entries. If present,
the value for this field must be non-zero for sections that have the SHF_ALLOC flag set. The
addional address space fields are only present in relocatable ELF object files. The fields are not
present in the absolute ELF file as generated by the linker.

The Section Header definition for relocatable object files:

```
 typedef struct {
        Elf32_Word     sh_name;
        Elf32_Word     sh_type;
        Elf32_Word     sh_flags;
        Elf32_Addr     sh_addr;
        Elf32_Off      sh_offset;
        Elf32_Word     sh_size;
        Elf32_Word     sh_link;
        Elf32_Word     sh_info;
        Elf32_Word     sh_addralign;
        Elf32_Word     sh_entsize;
        unsigned char  sh_addrspace;  // additional address space field
        unsigned char  sh_reserved[3];// reserved for future use
} Elf32_Shdr;
```

The Symbol Table Entry definition for relocatable object files:

```
typedef struct {
        Elf32_Word     st_name;
        Elf32_Addr     st_value;
        Elf32_Word     st_size;
        unsigned char  st_info;
        unsigned char  st_other;
        Elf32_Half     st_shndx;
        unsigned char  st_addrspace;  // additional address space field
        unsigned char  st_reserved[3];// reserved for future use
} Elf32_Sym;
```

The sh_reserved and st_reserved fields are required to pad to a 32 bit boundary.

The following address space values are defined:

Space	Value
bit	1
bita	2
iram	3
near	4
far	5
shuge	6
huge	7
code	8

1.4 Relocation Expression Stack

For those situations in which the relocation value cannot be expressed as a simple symbol value plus
an addend, there are three special relocation types (ELF32_R_TYPE) used to evaluate an arbitrary expression on a relocation stack. These relocation types are referred to as extended relocations. Other relocation types are ordinary relocations.

A relocation stack is a standard last-in-first-out data structure containing 32-bit values. A hosted environment must not place any arbitrary limit on the depth of the stack. An embedded environment may impose any limit on stack depth or omit the relocation stack entirely (effectively, a maximum stack depth of zero).

A target supporting the relocation expression stack must define the following relocation types in addition to the target specific relocation types:

Relocation type	Value
R_TASKING_PUSH	253
R_TASKING_OPER	254
R_TASKING_POP	255

R_TASKING_PUSH
This relocation type indicates that the sum of the symbol value (the value of symbol number zero is zero) plus the signed r_addend value should be pushed onto the relocation stack.

R_TASKING_OPER
This relocation type defines an operation to be performed on one or more stack values. The operation is specified by the sum of the symbol value (the value of symbol number zero is zero) plus the signed r_addend value. Operations are shown in Table 8. In the table, Stack 0 indicates the value on the top of the stack, and Stack 1 indicates the value one level beneath the top of the stack.

R_TASKING_POP
Indicates the end of a relocation expression. When the R_TASKING_POP operation is encountered, there should be exactly one value on the stack. This value, which is consumed by this operation, becomes the new relocation value for the ordinary relocation type specified in the R_TASKING_POP relocation. The relocation type is specified by the sum of the symbol value (the value of symbol number zero is zero) plus the signed r_addend value
It is the responsibility of the relocation engine to ensure that the stack is empty after a R_TASKING_POP, before an ordinary relocation, and after linking is complete. A sequence of relocations which causes a stack underflow does not conform to this specification.

The following Relocation Stack Operations are defined:

Relocation Value	Stack 0 Before Operation	Stack 1 Before Operation	Stack 0 After Operation	Operation
0	X		X	No operation
1	X		-X	Negation (2s complement)
2	X		~X	Bitwise NOT (1s complement)
3	X		!X	Boolean NOT (zero ->1, nonzero -> 0)
4	Y	X	X * Y	Multiplication
5	Y	X	X / Y	Division
6	Y	X	X % Y	Remainder
7	Y	X	X + Y	Addition
8	Y	X	X - Y	Subtraction
9	Y	X	X <<< Y	Logical shift left
10	Y	X	X >>> Y	Logical shift right
11	Y	X	X << Y	Arithmetic shift left
12	Y	X	X >> Y	Arithmetic shift right
13	Y	X	X < Y	1 if X < Y, otherwise 0
14	Y	X	X <= Y	1 if X <= Y, otherwise 0
15	Y	X	X > Y	1 if X > Y, otherwise 0
16	Y	X	X >= Y	1 if X >= Y, otherwise 0
17	Y	X	X == Y	1 if X equals Y, otherwise 0
18	Y	X	X != Y	1 if X does not equal Y, otherwise 0
19	Y	X	X & Y	Bitwise AND
20	Y	X	X | Y	Bitwise OR
21	Y	X	X || Y	Bitwise XOR
22	Y	X	X && Y	1 if X and Y both nonzero, otherwise 0
23	Y	X	X || Y	1 if X or Y or both nonzero, otherwise 0

Note that in most cases, the stack values are treated as unsigned. However, arithmetic shifts and logical shifts are treated differently.

Logical shift left:
Zeroes are shifted in on the right.

Logical shift right:
Zeroes are shifted in on the left.

Arithmetic shift left:
Zeroes are shifted in on the right, and the most significant bit is always unaffected. Arithmetic shift right: Copies of the most significant bit are shifted in on the left

2 DWARF Debug Information

The C166 tool chain uses DWARF for passing HLL debug information from the compiler to the debugger.

2.1 DWARF register mapping

DWARF represents register names effectively as small integers. These numbers are used in the OP_REG and OP_BASEREG atoms to locate values. The mapping of DWARF register numbers to the C166 register set is as follows.

Register	Atom	Ranges
Rn	a = n	0 < = n < = 15; 0 < = a < = 15
USR0	a	a = 288
SP	a	a = 289
MAC	a	a = 290
MAH	a	a = 291
MAL	a	a = 292
MAE	a	a = 293
MRW	a	a = 294
IDX0	a	a = 295
IDX1	a	a = 296
QX0	a	a = 297
QX1	a	a = 298
QR0	a	a = 299
QR1	a	a = 300
CF Info return_address_register	a	a = 301
IP	a	a = 302
CSP	a	a = 303
SPSEG	a	a = 304
DPP0	a	a = 305
DPP1	a	a = 306
DPP2	a	a = 307
DPP3	a	a = 308

Note: the "CF Info return_address_register" register value has been defined to prevent the number from being used for a regular register in the future. Otherwise debuggers could run into problems when reading older objects where the number used for the return_address_register in the call frame information would overlap with a regular register's number in newer objects. The "CF Info return_address_register" is a virtual register and it is not intended to show up in any DWARF expression.

2.2 Function Attributes

Function attributes describing the combination of memory model, stack model and other calling convention details, are conveyed with additional tool-chain specific values using the DWARF calling convention attribute DW_AT_calling_convention.

2.2.1 DWARF Function Calling Convention

Encoding	Symbolic Value	Meaning
0x01	DW_CC_normal	Huge function model, return address on system stack (default)
0x02	DW_CC_program	Not used (see DWARF 3 specification)
0x03	DW_CC_nocall	Not used (see DWARF 3 specification)
0x65	DW_CC_interrupt	Function is an interrupt handler, return address on system stack
0x66	DW_CC_near_system_stack	Near function model, return address on system stack
0x67	DW_CC_near_user_stack	Near function model, return address on user stack
0x68	DW_CC_huge_user_stack	Huge function model, return address on user stack

2.3 TASKING Type Qualifier Extensions

The additional C type qualifiers are specified using the DW_AT_address_class attribute.

2.3.1 Version 1

The values that will be used when compiler option --dwarf-encoding=1 is used:

Qualifier	Value	Remark
__bit	1	
__near	2	
__far	3	
__shuge	4	
__huge	5	
__code	6	not really used; is implicit for functions
__near32	7	Same as __near, but occupies 32-bit storage in memory/stack. (not usable in c-code, automatically assigned by the partitioner)

2.3.2 Version 2

The values that will be used when compiler option --dwarf-encoding=2 is used:

Qualifier	Value	Remark
__bit	1	
__bita	8	
__iram	9	
__near	2	
__near32	7	Same as __near, but occupies 32-bit storage in memory/stack. (not usable in c-code, automatically assigned by the partitioner)
__far	3	
__shuge	4	
__huge	5	
__code	6	not really used; is implicit for functions

2.4 __unaligned and __packed__

The tag DW_TAG_packed_type corresponds to the *_unaligned qualifier. The use of the __packed_* attribute is not encoded explicitly in the sense that for a struct with this attribute all members will have a DW_TAG_packed_type, as if they all had __unaligned attributes.

2.5 Call Frame Information

The following information should be read in conjunction with the definitions in Section 6.4 of the DWARF standard document.

2.5.1 Call Stack and Memory Models

The size and the save area of the return address differ across the various memory models. This has to be reflected by the debug info for the debugger to be able to walk up the stack.

2.5.1.1 Basic Facts

- Each stack word is 16 bits in size.
- Conceptually, the return address consists basically of CSP:IP, but for "near" functions only IP will be pushed on the stack, meaning that the callee's CSP value is the same as the caller's then.
- Whether the return address is pushed on the system stack or the user stack depends on several factors. See the tables below.
- CSP does not change for the duration of one function.

2.5.1.2 Pending Issues

- Functions where variable length arrays (VLA) are used, switch to using R8 as the frame pointer in order to access automatic variables, while R15 still acts as SP. However, R15 is changed based on run-time data, when resizing VLAs, which cannot be determined at compile time. Therefore in VLA situations R8 should be used in the CFA calculations.
- Infrequently the C compiler needs to save the PSW register to the system stack for a very short period of time, causing the SP register to change in value. These so-called stack deltas also need to be reflected in the call frame information.

2.5.1.3 Known Limitations

- When single-stepping individual instructions into a function call in a user-stack model application, the return address is pushed onto the user-stack using multiple instructions. For these instructions no call frame information is issued, causing call frame information to be insufficient for stack walking or saved register retrieval when halting anywhere in such a push sequence.

2.5.1.4 Near Functions, Return Address on System-Stack

The below applies to functions explicitly or implicitly qualified __near __nousm.

Saved value	Stack
Return address	SP stack
Local automatic variables	R15 stack
CPU registers	R15 stack
System Stack Layout	
+0	IP

2.5.1.5 Huge Functions, Return Address on System-Stack

The below applies to functions explicitly or implicitly qualified __huge __nousm.

Saved value	Stack

Return address	SP stack
Local automatic variables	R15 stack
CPU registers	R15 stack

System Stack Layout	
+2	CSP
+0	IP

2.5.1.6 Near Functions, Return Address on User-Stack

The below applies to functions explicitly or implicitly qualified __near __usm.

Saved value	Stack
Return address	R15 stack (IP only, i.e. 16 bits)
Local automatic variables	R15 stack
CPU registers	R15 stack

User Stack Layout	
+0	IP

2.5.1.7 Huge functions, Return Address on User-Stack

The below applies to functions explicitly or implicitly qualified __huge __usm.

Saved value	Stack
Return address	R15 stack
Local automatic variables	R15 stack
CPU registers	R15 stack

User Stack Layout	
+2	CSP
+0	IP

2.5.1.8 Interrupt Functions

The below applies to functions qualified __interrupt.

Saved value	Stack
Return address	SP stack
Local automatic variables	R15 stack
CPU registers	SP stack

System Stack Layout	
+4	PSW
+2	CSP
+0	IP

2.5.2 Self-containedness

The compiler generates the call frame information in such a way that no information from sections other than .debug_frame should be required to produce a stack trace. For example, it should not be necessary to look up DW_AT_calling_convention attributes.

2.5.3 Addresses

A crucial point is that everywhere that an address is the final value to be calculated or used to read from memory (e.g. as the operand of a DW_OP_deref), it *must* be assumed to be a 32-bit linear byte address. This differs from how the processor itself behaves, i.e. as will also be mentioned further below, effects like page addressing or the use of SPSEG are all made "explicit" in the debug information.

This makes the debug information more complex, but has the advantage of requiring far fewer target-dependent code in the debugger.

2.5.4 Definition of CFA

For each address within a function with debug information, there should be a DWARF rule defining the so-called canonical frame address (CFA). Depending on the type of the function, this CFA may be associated with either the system stack or the user stack. This document does *not* describe how the CFA is defined for a given function type, and this may in fact change in the future without notice.

This is not a problem because the CFA is merely an abstract concept that does little more than help compress the stack unwinding rule table. As long as debuggers use the DWARF information correctly, they should not need a definition of the CFA. However, it is defined here that the CFA rule can always be assumed to evaluate to a quantity that represents a 32-bit linear address. This is in line with what was stipulated in Section "Addresses" above.

It is also noted that the CFA may be referenced from a location expression via DW_OP_call_frame_cfa. A related point is that the CFA and the DW_AT_frame_base are often related, but they should not be equated.

2.5.5 Determining the Return Address

The return address is defined by means of a "virtual" register. (As shown in the tabe above, this is now register 301, but debugger implementers are advised to just use the number specified in the Common Information Entry (CIE).)

The return address can be calculated using the appropriate rule in the standard way. It is stipulated here that the calculated quantity is always a 32-bit linear address, *also* for near functions, where the actual value saved on the stack is only 16 bits wide. In other words, the width of this virtual register is 32 bits in the same way that the width of R0 is 16 bits. (This is in line with what was stipulated in Section "Addresses" above.) Knowing this width is necessary in particular when the rule for the return address defines a memory location (as opposed to a value): 32 bits must be read from that location.

When a function has debug information, but there is no rule for the return address register or it is explicitly DW_CFA_undefined, that function is the topmost function on the stack. (That is, stack unwinding should stop there.)

2.5.6 Determining Stack Pointer Register Values

The values of the system (SP) and user (R15) stack pointer registers in higher frames can be determined in exactly the same way as those of other registers. For example, an empty huge function will have a rule which states that the value of SP (289) in this frame's caller is this frame's SP value plus 4, i.e. the "stack delta" is 4. The same applies to R15.

Note that although SP and R15 are used as stack pointers, the values calculated from these rules should *not* be confused with the "abstract", 32-bit wide stack pointers. The rules simply yield 16-bit values in exactly the same way as they do for e.g. R0.

2.5.7 SPSEG

Some derivatives use SPSEG to determine which segment is used for the system stack, while other derivatives do not have this SPSEG register. This should be entirely transparent to the debugger implementer because SPSEG is referenced in the appropriate rules only where and when necessary.

2.5.8 Near Data Addresses

When R15 is used as a (stack) pointer by the processor itself (e.g. when pushing or popping), the processor uses one of four DPP registers to determine the linearized address. This page addressing is implicit then, but as already mentioned in Section "Addresses" above, in the call frame information it is encoded explicitly in the DWARF expression. That is, where necessary bits 14 and 15 of R15 are extracted to select a DPP register and this is combined with bits 0...13. This results in fairly long DWARF expressions (which even use branches), but it has the advantage of being entirely transparent to the debugger, as long as it can handle these complex DWARF expressions correctly.

3 Code compaction

3.1 Introduction

Using the tool chain's code compaction ("coco" or "reverse inlining") optimization results in debug information that still complies with the DWARF standard, but certain extra intelligence is required in the debugger to still provide a good debugging experience to the user.

When two otherwise unrelated pieces of code are sufficiently similar, the compiler generates a single "coco function", which will be invoked from the two original locations. In the debug information, these functions can be recognized by their DW_AT_name, which begins with ".cocofun_", followed by one or more digits. Note that nested code compaction is possible, i.e. .cocofun1(...) could in turn invoke .cocofun2(...).

As far as producing a call stack trace is concerned, nothing changes: coco functions have the same sort of stack-related debug information (Section 6.4 of the DWARF standard) as normal functions. It is recommended, however, that in its stack window the debugger does not show call frames belonging to coco functions, because these are meaningless to the user.

3.2 Line information

Correct use of the line information is more difficult. If a coco function occupies the address range 0x100-0x120, each instruction in this range will be associated with more than one source line. DWARF allows this, but typically, a debugger will then revert to showing disassembly, because it is unable to disambiguate the situation.

It should be noted that the DWARF info currently generated by the tool chain does not explicitly express the relationship between coco functions and their "parents". Consequently, a slightly heuristic method is required.

Consider the following example, where similar code from foo1() and foo2() has been extracted into a coco function.

```
0x100 .cocofun1: mov ... ; lines 11 and 21
0x102 rets

0x200 foo1: mov ...    ; line 10
0x202 calls .cocofun1 ; line 11
0x204 mov ...          ; line 12
0x206 rets

0x300 foo2: mov ...    ; line 20
0x302 calls .cocofun1 ; line 21
0x304 mov ...  ; line 22
0x306 rets
```

If the target halts at 0x100, the debugger determines (via the name) that this lies within a coco function and that, according to the line info, this could represent both source lines 11 and 21. It should then perform a stack trace, which indicates that the return address at top-of-stack is, say, 0x304. From the line information in that vicinity (20-22) it can conclude that, in this case, address 0x100 probably means line 21, not 11.

Note that other optimizations may lead to instruction re-ordering within foo2(). Therefore, it is recommended to not just check the return site (0x304), but also a few bytes before and after that.

3.3 Execution control

Taking into account coco requires only a few changes in the debugger regarding source-level stepping, as long as the "disambiguated" line number is determined correctly. Note, however, that when the next instruction is a call and the user does a "step over", it needs to be checked whether the callee is a coco function. If so, semantically that is the same function, so at instruction level a step into should be done then.

3.4 Variable access

When a coco function is generated, this may involve the parent functions' local variables. In the debug information, the TAG_subprogram associated with a coco function itself will not enclose TAG_variables then, however; they remain associated with the TAG_subprograms of their respective parents, among other reasons because otherwise their (semantic) scope would not be represented properly.

Firstly, this affects lookup by identifier. If, in the earlier example, the target halts at 0x100 and the user requests evaluation of the expression "a + 1", the debugger's lookup procedure needs to check the debug information of the first non-coco function on the stack, e.g. foo2(), not .cocofun1 itself.

However, the actual storage of the variable may be affected by the presence of a coco function. In general, the DW_AT_location attribute of the TAG_variable can refer to addresses that lie within the coco function, i.e. outside the parent function. In the above example, foo2's local variable 'a' could be stored in register R0 when in the range 0x304-0x306 and in R1 when in 0x100-0x102.