

XMOS XS1 32-Bit Application Binary Interface

Version 9.7



Publication Date: 2009/10/20

Copyright © 2009 XMOS Ltd. All Rights Reserved.

1 Introduction

The Executable and Linkable Format (ELF) [1] defines a linking interface for compiled programs. This document is the processor-specific supplement for use with ELF on 32-bit XCore XS1 processors [2]. It is intended for linking objects compiled from C [3], XC [4] and assembly code [5].

2 Execution Environment

The execution environment is a single XCore processor within a global shared memory system. The program image consists of a set of ELF loadable segments.

3 Types

The distinct XC/C data types are described in Figure 1. Sizes and alignments are given in bits. In addition:

- by default the `char` type is unsigned,
- `long` is the same as `int`,
- `long double` is the same as `double`,
- the underlying type of an enumerated type is `int`, and
- function pointers are the same as data pointers.

Figure 1: XC/C data types

Type	Size	Align	XC	C	Meaning
<code>char</code>	8	8	✓	✓	Character type
<code>short</code>	16	16	✓	✓	Short integer
<code>int</code>	32	32	✓	✓	Native integer
<code>long</code>	32	32	✗	✓	Long integer
<code>long long</code>	64	32	✗	✓	Long long integer
<code>float</code>	32	32	✗	✓	32-bit IEEE float
<code>double</code>	64	32	✗	✓	64-bit IEEE float
<code>void *</code>	32	32	✗	✓	Data pointer
<code>chanend</code>	32	32	✓	✗	Channel end resource identifier
<code>port</code>	32	32	✓	✗	Port resource identifier
<code>timer</code>	32	32	✓	✗	Timer resource identifier
<code>clock</code>	32	32	✓	✗	Clock resource identifier

Structure types pack according to the regular SY5V rules:

- field offsets are aligned according to the field's type,
- a structure is aligned according to its most aligned member, and
- tail padding is added to make the structure's size a multiple of its alignment.

3.1 Bit-Fields

The following declared types may be specified in a bit-field's declaration: `char`, `short`, `int`, `long` and `enum`.

If an enum type has negative values, enum bit-fields are signed. Otherwise enum bit-fields are unsigned. All other bit-field types are signed unless explicitly unsigned.

Bit-fields pack from the least significant end of the allocation unit. Each non-zero bit-field is allocated at the first available bit offset that allows the bit-field to be placed in a properly aligned container of the declared type. Non bit-field members are allocated at the first available offset satisfying their declared type's size and alignment constraints.

A zero-width bit-field forces padding until the next bit-offset aligned with the bit-field's declared type.

Unnamed bit-fields are allocated space in the same manner as named bit-fields.

A structure is aligned according to each of the bit-field's declared types in addition to the types of any other members. Both zero-width and unnamed bit-fields are taken into account when calculating a structure's alignment.

4 Function Calling

Function calling uses the first four registers to pass parameters. Additional parameters are passed on the stack.

Except where otherwise stated, data types with size greater than `int` and all structures are passed by passing a pointer. The callee must make a copy of the structure if it needs to be modified. Scalar types smaller than 32 bits are passed as zero or sign extended 32-bit values.

Objects of type `long long` and `double` are passed as if they consisted of two 32-bit values, the least significant half passed first.

Variadic arguments are treated the same as other functions, except that any bound parameters (see Appendix 4.2) are omitted.

4.1 Function Returns

Function returning uses the first four registers to pass return values. Additional return values are passed on the stack (caller workspace).

Except where otherwise stated, data types with size greater than `int` and all structures are returned as follows: for each such type in the return type list, the caller passes as an implicit parameter the address of the return destination. This must be a valid address to which the return value can be assigned. These parameters are placed before the formal parameters and are ordered by their types in the return list. Scalar types smaller than 32 bits are returned as zero or sign extended 32-bit values.

Objects of type `long long` and `double` are returned as if they consisted of two 32-bit values, the least significant half returned first.

4.2 Array Bound Parameters

If the size of the first dimension of an array is missing then this value is passed to the function as an implicit parameter. A list of implicit bound parameters is constructed from the formals (in the order that they appear) and is placed after the formals and any return parameters. For example:

```
void f(int x[][10], int y);  
  
register    r0  r1  r2  
parameter  &x  y   x.bound
```

If the size of the bound is unknown to the caller then `MAX_UINT` is passed. This ensures that any run-time array bound checks will not fail when an index is valid.

4.3 Transaction functions

Transactions functions take an implicit parameter which represents the state of the `last_out` variable described in Appendix 11.1. This parameter is passed before the formal parameters. The updated value of the `last_out` variable as returned as an implicit return value.

5 The `null` constant

The `null` constant is represented by the 32-bit value 0.

6 Global Arrays

Whenever the definition of a global array `a` is exported from a compilation unit, `a.globound` must be declared as a global absolute symbol and its value set to the number of elements of the first dimension of the array.

7 Register Assignments

The register assignments are described in Figure 2.

Figure 2: Register Assignments

Registers	Type	Usage
<i>r0-r3</i>	Caller save	Argument and return values
<i>r4-r10</i>	Callee save	Scratch
<i>r11</i>	Caller save	Scratch
<i>cp</i>	Callee save	Constant pointer
<i>dp</i>	Callee save	Data pointer
<i>sp</i>	Callee save	Stack pointer
<i>lr</i>	Callee save	Link register

- The *cp* register points to a constant pool (see Appendix 12).
- The *dp* register points to static data (see Appendix 13).
- The *sp* register holds the base address of the stack of the current function (see Appendix 8).
- The *lr* register holds the address to return to when a function completes (see Appendix 9).

8 Stack Frame

The diagram in Figure 3 illustrates the organisation of two adjacent stack frames. The outgoing arguments and the incoming returns are written in order so that earlier arguments have smaller *sp* offsets.

Figure 3: Stack Frame Layout

outgoing returns (callee writes)
incoming arguments (caller writes)
branch link
locals and temporaries
incoming returns (callee writes)
outgoing arguments (caller writes)
branch-link next (required only for non-leaves)

9 Function Entry and Exit

Before entering a function, the caller must guarantee that $sp[0]$ does not contain data that is required after the call. It copies into lr the address of the instruction to execute after the function returns. On exiting a function, the callee copies the on-entry value of the lr register to the pc register.

9.1 Example

A function f is called with the instruction `bl f`. This instruction saves the value of $pc+1$ to lr .

The entry sequence for f is `ent sp n` , where n is the size in words of the stack frame. This instruction saves lr to $sp[0]$ and extends the stack frame by n words.

The exit sequence for f is `ret sp n` . This instruction adjusts the stack pointer to its value prior to entry and loads $sp[0]$ to pc .

10 Select Function Entry and Exit

Before entering a select function, the caller must guarantee that $sp[0]$ does not contain data that is required after the call.

The XCore has a status register that contains an EEBL bit (set using the `set sr` instruction). If this bit is unset then the select function has been called in *execution mode*, otherwise it has been called in *enabling mode*. If a select function is called in execution mode then the entry and exit requirements are the same as in Appendix 9.

Before entering a select function in enabling mode, the caller must guarantee that $sp[0]$ does not contain data that is required after the call. It copies into lr the address of the instruction to execute after the function returns. Additionally, the last argument passed to the function must be an implicit parameter containing the address of the instruction to execute after an event is taken and the body of a corresponding case statement returns.

On exiting a select function in enabling mode, the callee copies the on-entry value of the lr register to the pc register. The value of the sp must not be decremented (it is not treated as callee save for this purpose). Any value stored to the stack must not be modified by the caller whilst in *enabling mode*.

11 Channel Communication

An object is communicated over a channel end as a stream of bytes. The number of bytes sent is equal to the size of the object. Bytes are sent in reverse order, so that the first byte sent is the byte that would have the highest address if the object was stored in memory.

For input and output statements on channel ends qualified `streaming` the object is communicated over the channel end directly. No additional control tokens are communicated.

The communication sequence for an input statement outside a transaction is defined to be the same as the communication sequence for a slave transaction consisting of just the input statement. The communication sequence for an output statement outside a transaction is defined to be the same as the communication sequence for a master transaction consisting of just the output statement.

11.1 Transactions

Transactions may only be used with unqualified channel ends. The communication sequence for each input and output enclosed in transactions depends on whether the previous operation on the channel in the transaction was an input or output. In the description below we represent this state with the variable `last_out`.

At the start of a master transaction the following actions are performed on the transactor:

1. Output an *end* control token
`outct res[c], CT_END`
2. Set `last_out` to 0

At the start of a slave transaction the following actions are performed on the transactor:

1. Check for an *end* control token
`checkct res[c], CT_END`
2. Set `last_out` to 1

For each input enclosed in a transaction *last_out* is set to 0. If this causes the value of *last_out* to change from 1 to 0 then an *end* control token is output. Finally the object is output as a stream of bytes.

For each output enclosed in a transaction *last_out* is set to 1. If this causes the value of *last_out* to change from 0 to 1 then an *end* control token is input. Finally the object is input as a stream of bytes.

At the end of a transaction if *last_out* is set to 1 the following actions are performed:

1. Output an *end* control token
outct res[c], CT_END
2. Check for an *end* control token
checkct res[c], CT_END

At the end of a transaction if *last_out* is set to 0 the following actions are performed:

1. Check for an *end* control token
checkct res[c], CT_END
2. Output an *end* control token
outct res[c], CT_END

12 Constant Pool

There is a single global constant pool, pointed to by the *cp* register. The linker adds the global symbol *_cp* and places all sections with the SHF_CP flag after this symbol. The *cp* register is initialised during the bootstrap process to the *_cp* symbol.

Named global read-only objects are placed in the *.cp.rodata* section and accessed via the *cp* register.

The compiler may place unnamed constants in mergeable sections. The *.cp.const4* section holds 4-byte mergeable constants. The *.cp.const8* section holds 8-byte mergeable constants. The *.cp.string* section holds unnamed string constants.

13 Data Region

There is a single global data pool, pointed to by the *dp* register. The linker adds the global symbol `_dp` and places all sections with the `SHF_DP` flag after this symbol. The *dp* register is initialised during the bootstrap process to the `_dp` symbol.

Writable named objects are placed in the `.dp.data` section or, if zero initialised, the `.dp.bss` section and are accessed via the *dp* register.

All global objects are word aligned. Unsigned scalar types smaller than 32 bits are stored as zero extended 32-bit values.

14 Clock Blocks

Before the program entry point is called, clock block 0 is configured to be clocked off the reference clock with no divide and is put into a running state. This clock block must not be reconfigured. The resource identifier for clock block 0 is given in `<xs1_user.h>` by the define `XS1_CLKBLK_REF`.

15 Processor-Specific Relocation Types

The processor-specific relocation types are listed in Figure 4. The relocation table is given in Figure 5.

Figure 4: Relocation Types

Field	Meaning
word32	Specifies a 32-bit field occupying 4 bytes.
ru6	Specifies an unsigned 6-bit field contained within 2 bytes. The value is placed in bits 0-5. (For example, <code>ldwdp</code> .)
lru6	Specifies an unsigned 16-bit field contained within 4 bytes. Least significant 6 bits are placed in bits 16-21. Most significant 10 bits are placed in bits 0-9. (For example, prefixed <code>ldwdp</code> .)
u10	Specifies an unsigned 10-bit field contained within 2 bytes. Value is placed in bits 0-9. (For example, <code>ldwcp1</code> .)
lu10	Specifies an unsigned 20-bit field contained within 4 bytes. Least significant 10 bits are placed in bits 16-25. Most significant 10 bits are placed in bits 0-9. (For example, prefixed <code>ldwcp1</code> .)
ru6s	Specifies a signed value encoded in 7 bits within 2 bytes. The magnitude is encoded as with <code>ru6</code> . Bit 10 is set to 0 for positive and 1 for negative numbers. (For example, <code>bu</code> .)
lru6s	Specifies a signed value encoded in 17 bits within 4 bytes. The magnitude is encoded as with <code>lru6</code> . Bit 26 is set to 0 for positive and 1 for negative numbers. (For example, prefixed <code>bu</code> .)
u10s	Specifies a signed value encoded in 11 bits within 2 bytes. The magnitude is encoded as with <code>u10</code> . Bit 10 is set to 0 for positive and 1 for negative numbers. (For example, <code>ldap</code> .)
lu10s	Specifies a signed value encoded in 21 bits within 4 bytes. The magnitude is encoded as with <code>lu10</code> . Bit 26 is set to 0 for positive and 1 for negative numbers. (For example, prefixed <code>ldap</code> .)

Figure 5: Relocation Table

Name	Value	Field	Calculation
R_XCORE1_NONE	0	none	none
R_XCORE1_ADDR32	1	word32	$S + A$
R_XCORE1_DP_REL6	2	ru6	$(S + A - dp)/4$
R_XCORE1_DP_REL16	3	lru6	$(S + A - dp)/4$
R_XCORE1_CP_REL6	4	ru6	$(S + A - cp)/4$
R_XCORE1_CP_REL16	5	lru6	$(S + A - cp)/4$
R_XCORE1_CP_REL10	6	u10	$(S + A - cp)/4$
R_XCORE1_CP_REL20	7	lu10	$(S + A - cp)/4$
R_XCORE1_REL6	8	ru6s	$(S + A - P)/2$
R_XCORE1_REL16	9	lru6s	$(S + A - P)/2$
R_XCORE1_REL10	10	u10s	$(S + A - P)/2$
R_XCORE1_REL20	11	lu10s	$(S + A - P)/2$
R_XCORE1_LSIZE20	12	lu6	$S + A$

Key:

- A* The addend used to compute the value of the field.
- P* The place (section offset or address) of the storage unit. being relocated (computed using *r_offset*).
- S* The value of the symbol whose index resides. in the relocation entry.
- dp* The value of the symbol *_dp*.
- cp* The value of the symbol *_cp*.

An error must be issued if the computed value does not fit in the allocated bits. For calculations using word or short offsets, an error must be issued on misalignment. Relocation information for a section is normally placed in a section called `.rel` followed by the name of the section to which the relocations apply (or `.rela` if addends are used). For example, relocation information for `.text` is placed in `.text.rel` or `.text.rela`.

16 Sections

The ABI defined sections for a linkable object are given in Figure 6.

Figure 6: ABI sections

Name	Type	Attributes SHF_ALLOC +	Size (bytes)	Description
.text	@progbits	SHF_EXECINSTR	none	program code
.dp.data	@progbits	SHF_WRITE + SHF_DP	none	initialised data
.dp.rodata	@progbits	SHF_WRITE + SHF_DP	none	read only data
.cp.rodata	@progbits	SHF_CP	none	read only data
.dp.bss	@nobits	SHF_WRITE + SHF_DP	none	zero-initialised data
.cp.string	@progbits	SHF_MERGE + SHF_STRINGS + SHF_CP	1	mergeable strings
.cp.const4	@progbits	SHF_MERGE + SHF_CP	4	mergeable constants
.cp.const8	@progbits	SHF_MERGE + SHF_CP	8	mergeable constants

17 Processor Specific Extensions

Two additional sections are defined: an expression section and a type section. These sections allow information about a compiled program to be communicated to the linker.

17.1 Expression Section

The value of a symbol that depends upon symbols in other objects can be expressed as an expression that is resolved during linkage. These expressions can contain constants, strings (from the string table) and symbol values.

The expression section has type SHT_EXPR with the name `.expr`. A single expression section is allowed. It contains an array of structures of the format:

```
typedef struct {
    Elf32_Word type;
    Elf32_Word result;
    Elf32_Word op1;
    Elf32_Word op2;
    Elf32_Word op3;
} Elf32_Expr;
```

The `result` field is a symbol number which denotes where the result is stored. The `type` field determines the arithmetic operation and whether each of the three operands is a constant value or an index into the symbol table as described in the table in [Figure 7](#).

Figure 7: Expression section entries and their meanings

Bits	Value	Meaning
0-1	0	Operand 1 is a constant.
	1	Operand 1 is a symbol index.
	2	Operand 1 is a string index.
2-3	0	Operand 2 is a constant.
	1	Operand 2 is a symbol index.
	2	Operand 2 is a string index.
4-5	0	Operand 3 is a constant.
	1	Operand 3 is a symbol index.
	2	Operand 3 is a string index.
6-14	0	Operator is NULL (unused.)
	1	Specifies that <i>result</i> is $op1 + op2$.
	2	Specifies that <i>result</i> is $max(op1, op2)$.
	3	Specifies that <i>result</i> is $op1 * op2$.
	4	Specifies that <i>result</i> is $op1 - op2$.
	5	Specifies that if <i>op2</i> evaluates to 0 the string <i>op1</i> must be printed as a warning (if <i>op3</i> is 0 as an error).
	6	Specifies that <i>result</i> is the value of <i>op1</i> aligned up to size <i>op2</i> .
	7	Specifies that <i>result</i> is $op1 < op2$ (boolean).
	8	Specifies that <i>result</i> is $op1 > op2$ (boolean).
	9	Specifies that <i>result</i> is $op1 \leq op2$ (boolean).
	10	Specifies that <i>result</i> is $op1 \geq op2$ (boolean).
	12	Specifies that function <i>result</i> calls function <i>op1</i> .
	13	Specifies that the global <i>op1</i> is passed by reference to function <i>result</i> ; <i>op2</i> is a string which identifies a source line.
	15	Specifies that function <i>result</i> reads global variable <i>op1</i> ; <i>op2</i> is a string which identifies a source line.
	16	Specifies that function <i>result</i> writes global variable <i>op1</i> ; <i>op2</i> is a string which identifies a source line.
17	Specifies that <i>op1</i> holds the stack usage of function <i>result</i> .	
19	Specifies that <i>op1</i> contains the thread usage of function <i>result</i> .	
20	Specifies that <i>op1</i> contains the timer usage of function <i>result</i> .	
21	Specifies that <i>op1</i> contains the channel end usage of <i>result</i> .	
23	<i>op1</i> is a boolean value which is non-zero if function <i>result</i> and its callees have no side effects.	
24	<i>op1</i> is a boolean value which is non-zero if function <i>result</i> (but not necessarily its callees) have no side effects.	
25	Functions <i>result</i> and <i>op1</i> may be called in parallel; <i>op2</i> is a string which identifies a source line.	
27	Specifies that the user-visible name of symbol <i>result</i> is string <i>op2</i> .	

A symbol used as an operand may itself be the result of an expression. Any cyclic dependencies must be detected and either reported as errors or result in implementation-defined values being used.

Symbols that are the subject of linker expressions are of ELF type absolute. Until they are resolved they have value zero.

17.2 Type Information

The type of a variable can be encoded as a string and added as an entry to the type section. The type section has type SHT_TYPEINFO with name `.typeinfo`. For sections of this type, the `sh_link` field holds the section header index of the symbol table to which the type information applies. The `sh_info` field holds the value 0. A single type section is allowed. It contains an array of structures of the format:

```
typedef struct {
    Elf32_Word ti_symbol;
    Elf32_Word ti_type;
} Elf32_TypeInfo;
```

The `ti_symbol` member holds an index into the object file's symbol string table which holds the ASCII character representation of the symbol name. The `ti_type` member holds an index into the object file's symbol string table, which holds the character representation of the type string. If there is no type string associated with a symbol there should be no entry in the table.

The ASCII string encoding for the XC and C types is given in Figure 8.

Figure 8: Type Section

Type	XC	C	Encoding
signed char	✓	✓	sc
unsigned char	✓	✓	uc
signed short	✓	✓	ss
unsigned short	✓	✓	us
signed int	✓	✓	si
unsigned int	✓	✓	ui
signed long	✓	✓	sl
unsigned long	✓	✓	ul
signed long long	✗	✓	sll
unsigned long long	✗	✓	ull
float	✗	✓	ft
double	✗	✓	d
_Bool	✗	✓	b
long double	✗	✓	ld
t:n	✗	✓	$b(n:E(t))^a$
chan	✓	✗	ch
chanend	✓	✗	chd
port	✓	✗	p
port:n	✓	✗	p:n
timer	✓	✗	t
clock	✓	✗	ck
void	✗	✓	0
struct tag {t ₁ ,t ₂ }	✓	✓	s(tag){E(t ₁),E(t ₂)}
struct tag	✓	✓	s(tag){} ^b
union tag {t ₁ ,t ₂ }	✓	✓	u(tag){E(t ₁),E(t ₂)} ^c
union tag	✓	✓	u(tag){} ^d
enum {m ₁ ,m ₂ }	✓	✓	e(tag){E(m ₁),e(m ₂)}
t name	✓	✓	m(name){E(t)} ^e
t name = val	✓	✓	m(name){val} ^f

^abit-field member^bincomplete struct^cmembers subject to ordering rules^dincomplete union^estruct or union member^fenum member

Figure 9: Type Section (continued)

Type	XC	C	Encoding
void f(void)	✓	✓	f{0}(0)
transaction f(void)	✓	✗	ft{0}(0)
select f(void)	✓	✗	fs{0}(0)
void f()	✓	✓	f{0}() ^a
void f(t)	✓	✓	f{0}(E(t))
void f(t, ...)	✗	✓	f{0}(E(t), va)
void f(t ₁ , t ₂)	✓	✓	f{0}(E(t ₁), E(t ₂))
t ₂ f(t ₁)	✓	✓	f{E(t ₂)}(E(t ₁))
{t ₂ , t ₃ } f(t ₁)	✓	✓	f{E(t ₂), E(t ₃)}(E(t ₁))
*t	✗	✓	p(E(t))
&t	✗	✓	&(E(t))
t[n]	✓	✓	a(n:E(t))
t[n][m]	✓	✓	a(n:a(m:E(t)))
t[]	✓	✗	a(:E(t)) ^b
t[]	✓	✓	a(*:E(t)) ^c

^aincomplete in C

^ban array parameter with no size specified

^ca global extern array with no size specified

Type qualifiers appear alphabetically, followed by a colon and then the type encoding. If there are no type qualifiers then no colon is emitted before the type. The encodings are given in Figure 10. For example, `volatile const int` is encoded as `cv:si`.

Figure 10: Qualifiers

Qualifier	XC	C	Encoding
buffered	✓	✗	b
const	✓	✓	c
in	✓	✗	i
inline	✓	✓	(ignored)
?	✓	✗	n
out	✓	✓	o
register	✓	✓	(ignored)
restrict	✗	✓	r
volatile	✗	✓	v
void	✓	✓	w

The value of enum members must always be encoded, regardless of whether an explicit value is provided. The enum values are encoded as decimal values. Enum

members are ordered alphabetically by their name.

Named union members are ordered alphabetically by their name. Unnamed union bit-field members are placed after named members and ordered firstly according to their size (low to high), then alphabetically by the type string of the bit-field's type.

In C [3, Appendix 6.7.5.3] when a function parameter is declared with a qualified type, the qualifiers are discarded when emitting the type string. In XC the `const` and `volatile` qualifiers are discarded from function parameters which are not a reference type.

C structures and unions may introduce cycles due to forward declarations of struct and union types. Wherever cycles exist, the type of the structure or union that completes the cycle is encoded as an incomplete struct or union. For example, in the declaration

```
struct tag { struct tag *next; } foo;
```

the type of `foo` is encoded as

```
s(tag){p(s(tag){})}.
```

17.3 Other Symbols

For each function f defined and added to the symbol table, the following symbols may also be defined. The definition of one of these symbols must be accompanied by an SHT_EXPR entry as described in Figure 7.

Figure 11: Special symbols that may be defined for a function and their SHT_EXPR entries (see Figure 7)

Symbol for function f	Type Value (Bits 6-14)
$f.nstackwords$	17
$f.maxthreads$	19
$f.maxtimers$	20
$f.maxchanends$	21
$f.actnoside$	23
$f.locnoside$	24
$f.actnochandec$	30
$f.locnochandec$	31

Bibliography

- [1] The Santa Cruz Operation. System V Application Binary Interface, Edition 4.1. Website, 1997. <http://www.caldera.com/developers/devspecs/gabi41.pdf>.
- [2] David May. *The XNOS XS1 Architecture*. XNOS Limited, 2009. http://www.xmos.com/published/xs1_en.
- [3] International Organization for Standardization. *ISO/IEC 9899:1999: Programming Languages* — C. Wiley, West Sussex, England, December 1999.
- [4] Douglas Watt. *Programming XC on XNOS Devices*. XNOS Limited, Sep 2009. http://www.xmos.com/published/xc_en.
- [5] Douglas Watt. XS1 Assembly Language Manual (8.7). Website, 2008. <http://www.xmos.com/published/xas87>.

Disclaimer

XNOS Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. XNOS Ltd. makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.