Tools developer's guide

Document Number: XM000140A

Publication Date: 2015/4/21

XMOS®

Table of Contents

1 Introduction

IN THIS CHAPTER

- xTIMEcomposer development tools
- How to read this guide

The XMOS architecture enables a combination of interface, digital signal processing and control functions to be performed in software. An XMOS device consists of one or more xCORE tiles, each comprising a multicore microcontroller with tightly integrated I/O and on-chip memory. Each xCORE tile has hardware support for executing several logical cores concurrently and has dedicated instructions for input and output.

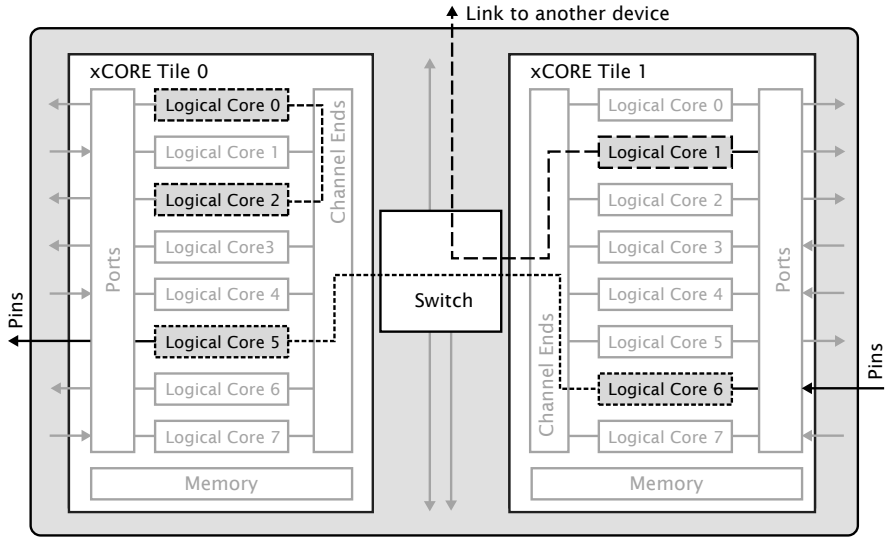

Figure 1: XMOS Architecture

The timing of instruction execution is deterministic, with each logical core guaranteed a slice of the processing. The logical cores can execute computations, handle real-time I/O operations and respond to multiple events. The I/O pins can be sampled or driven using a single instruction, and data rates can be controlled using timers or clocks. A high-performance switch enables communication between xCORE tiles and makes it easy to construct systems from multiple devices.

1.1 xTIMEcomposer development tools

The xTIMEcomposer development tools support a standard embedded development flow and are built on industry-standard platforms. The tools let designers use C, C++, or xC to program devices, or assembly if absolute control is required. Programs are typically debugged using the XMOS port of the GNU Debugger (GDB). A static timing analyzer can be used to validate that all real-time requirements are met for the target device at compile time, helping designers close timing without the need for complex test benches.

Board utilities are provided for loading programs onto hardware. During development, programs are typically loaded from a host PC over JTAG. In a manufactured design, programs are typically loaded from flash memory. The tools can encrypt programs on flash and can burn a secure bootloader and keys into on-chip OTP memory, ensuring program and device authenticity. The tools also support in-field upgrades, allowing multiple firmware releases to be managed over the product life cycle.

The formats and conventions used by the xTIMEcomposer tools form a well-defined interface for passing data between programs. By conforming to this interface, third-party developers are guaranteed that their tools are compatible with the xTIMEcomposer tools.

1.2 How to read this guide

This guide is divided into three sections of reference material:

- **Chapter 2** documents the XMOS application binary interface (ABI), which specifies how to produce code which is compatible with the libraries supplied with the xTIMEcomposer tools and with code produced by the XMOS compiler collection.
- **Chapter 3** documents the XMOS executable file format, which specifies how to produce executables that can be used with tools such as XSIM, XGDB, XFLASH and XBURN.
- **Chapter 4** documents the XMOS system call interface, which specifies the mechanism by which system services used by tools, such as debuggers and simulators, are interfaced on the host development system.

2 XMOS 32-bit application binary interface

IN THIS CHAPTER

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 multicore microcontrollers.[2] It is intended for linking objects compiled from xC,[3] C/C++ and assembly code.[4]

The following sections detail general ABI requirements for xCORE targets; specific requirements for XS1 and XS2 architectures are detailed in §2.16 and §2.17.

2.1 Execution environment

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

[1] System V Application Binary Interface, Edition 4.1. http://www.caldera.com/developers/devspecs/gabi41.pdf.
[2] The XMOS XS1 Architecture. http://www.xmos.com/published/xs1_en.
[3] Programming XC on XMOS Devices. http://www.xmos.com/published/xc_en.
[4] The XMOS xTIMEcomposer User Guide. http://www.xmos.com/published/xtools_en.

2.2 Types

The distinct xC/C data types are described in Figure 2. 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`,
- function pointers are the same as data pointers.

Type	Size	Alignment	xC	ANSI C	Meaning
`char`	8	8	Y	Y	Character type
`short`	16	16	Y	Y	Short integer
`int`	32	32	Y	Y	Native integer
`long`	32	32	Y	Y	Long integer
`long long`	64	*	Y	Y	Long long integer
`float`	32	32	Y	Y	32-bit IEEE float
`double`	64	*	Y	Y	64-bit IEEE float
`void`	32	32	N	Y	Data pointer
`chanend`	32	32	Y	N	Channel end resource identifier
`port`	32	32	Y	N	Port resource identifier
`timer`	32	32	Y	N	Software timer
`hwtimer_t`	32	32	Y	N	Timer resource identifier
`clock`	32	32	Y	N	Clock resource identifier

Key:
* Target dependent.

Figure 2: Data Types

Structure types pack according to the regular SYSV rules:

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

2.2.1 Enumerated types

Each enumerated type is associated with an underlying integer type that determines its size and alignment. If the enumerated type has a negative enumeration constant the underlying type is the first type in the following list in which all its enumeration constants can be represented: `int`, `long` and `long long`. Otherwise, the underlying type is the first type in the following list in which its the enumeration constants can be represented: `unsigned int`, `unsigned long` and `unsigned long long`.

2.2.2 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 marked as being 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.

2.3 Function calling

Function calling uses the first four registers (*r0* to *r3*) to pass parameters. Additional parameters are passed on the stack.

Aggregates and data types with a size greater than an `int` are passed using a pointer, with the exceptions:

- that objects of type `long long` and `double` are passed as if they consisted of two 32-bit values, the least significant half passed first;
- that the passing of aggregates is target dependent (see §2.16 and §2.17).

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.

Variadic arguments are treated the same as other functions, except that any bound parameters (see §2.3.2) are omitted.

2.3.1 Function returns

Function returning uses the first four registers to pass return values. Additional return values are passed on the stack, in the caller's frame (see §2.7).

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 address must be valid and must not alias the storage location of any object visible

to the callee. Implicit structure return 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.

2.3.2 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 are placed after the formals and any return parameters. For example, for the function:

```
void f(int x[][10], int y);
```

the parameters are passed in registers as follows:

register	*r0*	*r1*	*r2*
parameter	`&x`	`y`	`x.bound`

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

2.3.3 Transaction functions

Transactions functions take an implicit parameter which represents the state of the *last_out* variable described in §2.9.1. This parameter is passed before the formal parameters. The updated value of the *last_out* variable as returned as an implicit return value.

2.3.4 Select functions

A select function provides multiple entry points. The name of the symbol for the first entry point is the same as the name of the function. This entry point is used when the select function is called outside of a select statement. In this context, the call to the select function uses the same calling convention as a function taking the same list of parameters and returning `void`.

The name of the symbol for the second entry point is the name of the function with the suffix `.enable`. This entry point is used when the select function is called from a select statement and is called during the event enable phase of the select. When called via this entry point, the select function must configure the resources that appear in cases of the select function so they generate events when ready. In addition, each resource should be configured so that execution jumps to an event handler this executes the corresponding select case body if the event is taken. After configuring all its resources the select function shall return.

After the resources are configured the function containing the top-level select statement shall enable events. This may result in execution to jumping to an event handler. This is considered to be a indirect call a function taking no arguments having no return values that respects the standard register assignment conventions (see Figure 3). The event handler must handle the event before returning a second time by jumping to the value the link register had when the event was taken.

Select functions may need to store state during the event enable phase that can accessed by that select function's event handlers. The function containing the top-level select statement must allocate space for this state. A pointer to the start of this memory is passed as an implicit argument to the first select function called in the event enable phase. The alignment of this pointer is target dependent (see §2.16 and §2.17). This implicit argument is the first parameter of that function with the subsequent parameters being the same as the select function itself. The select function will store state to the start of the area of pointed to by this implicit argument and return the next aligned word of space after saved state as an implicit return value. If multiple select functions are called, the return value of the previous select function is passed as as the implicit argument to the next select function.

To determine the amount of space required to store saved state each select function must define an absolute symbol whose value is the number of words of state saved by that select function and all its callees. The name of the symbol is the name of the function with the suffix `.selectstatesavewords`.

2.4 The `null` constant

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

2.5 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.

2.6 Register assignments

The register assignments are described in Figure 3.

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

Figure 3: Register assignments

- The *cp* register points to a constant pool (see §2.10).
- The *dp* register points to static data (see §2.11).
- The *sp* register holds the base address of the stack of the current function (see §2.7).
- The *lr* register holds the address to return to when a function completes (see §2.8).

2.7 Stack frame

Figure 4 illustrates the organization of two adjacent stack frames. The outgoing arguments and the incoming returns are written in order so that earlier arguments have smaller *sp* offsets.

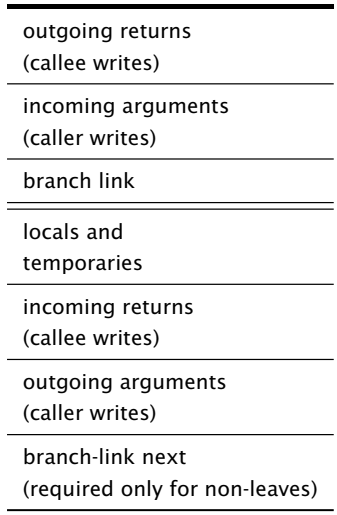

Figure 4: Stack Frame Layout

2.8 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* to the *pc* register.

2.8.1 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:

```
entsp 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:

```
retsp n
```

This instruction adjusts the stack pointer to its value prior to entry and loads *sp*[0] to *pc*.

2.9 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.

2.9.1 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 a input or output. In the description below this state is represented 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
   ```

2.10 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 initialized 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.

2.11 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 initialized during the bootstrap process to the `_dp` symbol.

Writable named objects are placed in the `.dp.data` section or, if zero initialized, 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.

2.12 Clock blocks

Before the program entry point is called, a clock block is configured to be clocked off the reference clock with no divide and is put into a running state. The global symbol `_default_clkblk` is set to the resource identifier of this block block. This clock block must not be reconfigured.

2.13 Processor-specific relocation types

The processor-specific relocation types are listed in Figure 5. The relocation table is given in Figure 6. Two fields are taken from the DWARF 3 standard.[5]

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`.

[5] DWARF Debugging Information Format Version 3. http://dwarfstd.org/doc/Dwarf3.pdf.

Field	Meaning
data32	Specifies a 32-bit field occupying 4 bytes.
data16	Specifies a 16-bit field occupying 2 bytes.
data8	Specifies a 8-bit field occupying 1 byte.
uleb32	Specifies a 32-bit field encoded in 5 bytes. The value is first encoded as an unsigned LEB128 number, as described in the DWARF 3 standard. Zero bytes are appended to pad the size to 5 bytes. The most significant bit of bytes 0, 1, 2 and 3 is set to 1.
sleb32	Specifies a 32-bit field encoded in 5 bytes. The value is first encoded as a signed LEB128 number, as described in the DWARF 3 standard. Zero bytes are appended to pad the size to 5 bytes. The most significant bit of bytes 0, 1, 2 and 3 is set to 1.
u6	Specifies an unsigned 6-bit field contained within 2 bytes. The value is placed in bits 0-5. (For example, ldwdp.)
lu6	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 ldwdp.)
u10	Specifies an unsigned 10-bit field contained within 2 bytes. Value is placed in bits 0-9. (For example, ldwcpl.)
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 ldwcpl.)
u6s	Specifies a signed value encoded in 7 bits within 2 bytes. The magnitude is encoded as with u6. Bit 10 is set to 0 for positive and 1 for negative numbers. (For example, bu.)
u6s	Specifies a signed value encoded in 17 bits within 4 bytes. The magnitude is encoded as with lu6. Bit 26 is set to 0 for positive and 1 for negative numbers. (For example, prefixed bu.)
u10s	Specifies a signed value encoded in 11 bits within 2 bytes. The magnitude is encoded as with u10. Bit 10 is set to 0 for positive and 1 for negative numbers. (For example, ldap.)
lu10s	Specifies a signed value encoded in 21 bits within 4 bytes. The magnitude is encoded as with lu10. Bit 26 is set to 0 for positive and 1 for negative numbers. (For example, prefixed ldap.)

Figure 5: Relocation types

Name	Value	Field	Calculation
R_XCORE1_NONE	0	none	none
R_XCORE1_DATA32	1	data32	*S + A*
R_XCORE1_DP_REL6	2	u6	*(S + A - dp) / 4*
R_XCORE1_DP_REL16	3	lu6	*(S + A - dp) / 4*
R_XCORE1_CP_REL6	4	u6	*(S + A - cp) / 4*
R_XCORE1_CP_REL16	5	lu6	*(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	u6s	*(S + A - P) / 2*
R_XCORE1_REL16	9	lu6s	*(S + A - P) / 2*
R_XCORE1_REL10	10	u10s	*(S + A - P) / 2*
R_XCORE1_REL20	11	lu10s	*(S + A - P) / 2*
R_XCORE1_ABS16	12	lu6	*S + A*
R_XCORE1_ULEB32	13	uleb32	*S + A*
R_XCORE1_DATA8	14	data8	*S + A*
R_XCORE1_DATA16	15	data16	*S + A*
R_XCORE1_ABS6	16	u6	*S + A*
R_XCORE1_SLEB32	17	sleb32	*S + A*
R_XCORE1_REL6_4	16	u6s	*(S + A - P) / 4*
R_XCORE1_REL16_4	17	lu6s	*(S + A - P) / 4*
R_XCORE1_REL10_4	18	u10s	*(S + A - P) / 4*
R_XCORE1_REL20_4	19	lu10s	*(S + A - P) / 4*

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`.

Figure 6: Relocation table

2.14 Sections

The ABI-defined sections for a linkable object are given in Figure 7.

Name	Type	Attributes SHF_ALLOC +	Size (bytes)	Description
.text	@progbits	SHF_EXECINSTR	none	Program code
.dp.data	@progbits	SHF_WRITE + SHF_DP	none	Initialized 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-initialized 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

Figure 7: ABI sections

2.15 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.

2.15.1 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 and 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 thee operands is a constant value or an index into the symbol table as described in the table in Figure 8. A single expression section is allowed.

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 the value zero.

2.15.2 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 in turn 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 ANSI C types is given in Figure 9.

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. For example, `volatile const int` is encoded as `cv:si`. The qualifier encodings are given in Figure 11.

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 ANSI C, 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.

ANSI 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){}) }
```

2.15.3 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 12.

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

Figure 8: Expression section entries and their meanings

Type	xC	C	Encoding
`signed char`	Y	Y	`sc`
`unsigned char`	Y	Y	`uc`
`signed short`	Y	Y	`ss`
`unsigned short`	Y	Y	`us`
`signed int`	Y	Y	`si`
`unsigned int`	Y	Y	`ui`
`signed long`	Y	Y	`sl`
`unsigned long`	Y	Y	`ul`
`signed long long`	N	Y	`sll`
`unsigned long long`	N	Y	`ull`
`float`	N	Y	`ft`
`double`	N	Y	`d`
`_Bool`	N	Y	`b`
`long double`	N	Y	`ld`
`t:n`	N	Y	`b(n:`$E(t)$`)`[a]
`chanend`	Y	N	`chd`
`port`	Y	N	`p`
`port:`n	Y	N	`p:`n
`timer`	Y	N	`swt`
`hwtimer_t`	Y	N	`t`
`clock`	Y	N	`ck`
`tileref`	Y	N	`cr`
`void`	N	Y	`0`
struct tag { t_1;t_2}	Y	Y	`s(tag){`$E(t_1)$`,`$E(t_2)$`}`
`struct tag`	Y	Y	`s(tag){}`[b]
union tag { t_1;t_2}	Y	Y	`u(tag){`$E(t_1)$`,`$E(t_2)$`}`[c]
`union tag`	Y	Y	`u(tag){}`[d]
enum tag { m_1,m_2}	Y	Y	`e(tag){`$E(m_1)$`,`$E(m_2)$`}`
`t name`	Y	Y	`m(name){`$E(t)$`}`[e]
client interface tag { m_1;m_2}	Y	N	`ic(tag)[s:]{`$E(m_1)$`,`$E(m_2)$`}`
server interface tag { m_1;m_2}	Y	N	`is(tag)[s:]{`$E(m_1)$`,`$E(m_2)$`}`
`t name`	Y	Y	`m(name){`$E(t)$`}`[e]
`t name = val`	Y	Y	`m(name){val}`[f]

[a] bit-field member
[b] incomplete struct
[c] members subject to ordering rules
[d] incomplete union
[e] struct or union member
[f] enum member

Figure 9: Type section

Type	xC	C	Encoding
`void f(void)`	Y	Y	`f{0}(0)`
`transaction f(void)`	Y	N	`ft{0}(0)`
`select f(void)`	Y	N	`fs{0}(0)`
`void f()`	Y	Y	`f{0}()`[g]
`void f(t)`	Y	Y	`f{0}(`*`E`*`(t))`
`void f(t,...)`	N	Y	`f{0}(`*`E`*`(t),va)`
void f(t_1,t_2)	Y	Y	`f{0}(`*`E`*`(`t_1`),`*`E`*`(`t_2`))`
t_2 f(t_1)	Y	Y	`f{`*`E`*`(`t_2`)}(`*`E`*`(`t_1`))`
{ t_2, t_3} f(t_1)	Y	Y	`f{`*`E`*`(`t_2`),`*`E`*`(`t_3`)}(`*`E`*`(`t_1`))`
`*t`	N	Y	`p(`*`E`*`(t))`[h]
`*t`	Y	N	`q(`*`E`*`(t))`[i]
`&t`	N	Y	`&(`*`E`*`(t))`
`t[n]`	Y	Y	`a(n:`*`E`*`(t))`
`t[n][m]`	Y	Y	`a(n:a(m:`*`E`*`(t)))`
`t[]`	Y	N	`a(:`*`E`*`(t))`[j]
`t[]`	Y	Y	`a(:`*`E`*`(t))`[k]
`t[n]`	Y	N	`a(!`*`offset`*`(n):`*`E`*`(t))`[l]

[g] incomplete in C

[h] C pointer

[i] xC pointer

[j] an array parameter with no size specified

[k] a global extern array with no size specified

[l] an array parameter with size specified as another parameter. The offset is the number of the size parameter minus the number of the array parameter.

Figure 10: Type section (continued)

Qualifier	xC	C	Encoding
alias	Y	N	a
buffered	Y	N	b
const	Y	Y	c
in	Y	N	i
[[clears_notification]]	Y	N	l
streaming	Y	N	m
?	Y	N	n
out	Y	N	o
restrict	N	Y	r
slave	Y	N	s
[[notification]]	Y	N	t
unsafe	Y	N	u
volatile	N	Y	v
void	Y	N	w
movable	Y	N	x
[[combinable]]	Y	N	k
[[distributable]]	Y	N	d
static	Y	N	e
inline	Y	Y	(ignored)
register	Y	Y	(ignored)

Figure 11: Qualifier encodings

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

Figure 12: Special symbols that may be defined for a function and their SHT_EXPR entries

2.16 XS1-specific requirements

The following sections detail the specific ABI requirements for XS1 targets.

2.16.1 Types

`long long` and `double` are 32-bit aligned.

2.16.2 Functions

All functions are 16-bit aligned.

The stack pointer must be 32-bit aligned on entry to a function.

For select functions, the caller must ensure that the initial implicit state pointer argument that is passed to the enable function is 32-bit aligned. The enable function must ensure that the returned state pointer is 32-bit aligned.

2.16.3 Constant pool/data region

All global objects are 32-bit aligned.

2.17 XS2-specific requirements

The following sections detail the specific ABI requirements for XS2 targets.

2.17.1 Types

`long long` and `double` are 64-bit aligned.

2.17.2 Functions

All functions are 32-bit aligned and must not assume anything about the current issue mode on function entry.

The stack pointer must be 64-bit aligned on entry to a function.

An aggregate containing a single member is passed as if the function took an argument of that member type. This rule applies recursively.

An aggregate containing a single member is returned as if the function returned an argument of that member type. This rule applies recursively.

For select functions, the caller must ensure that the initial implicit state pointer argument that is passed to the enable function is 64-bit aligned. The enable function must ensure that the returned state pointer is 64-bit aligned.

2.17.3 Constant pool/data region

All global objects are aligned by taking the maximum alignment determined by the following rules:

- global objects are aligned according to their type;
- arrays are 64-bit aligned;
- aggregate types with a size of at least 8 bytes are 64-bit aligned;
- all global objects are 32-bit aligned.

2.17.4 Sections

Global objects with 32-bit alignment must be placed in the following sections, separate to global objects with 64-bit alignment:

- `.dp.data.4`
- `.dp.rodata.4`
- `.cp.rodata.4`
- `.dp.bss.4`

The existing section names will continue to be used for 64-bit-aligned data.

2.17.5 Identifying dual issue code

To help identify dual-issue code, symbols of the following form must be added to the symbol table:

Name	Description
`$s.`<number>	Start of a sequence of single issue instructions
`$d.`<number>	Start of a sequence of dual issue instructions

These symbols can be used by disassemblers to identify whether a sequences of bytes in executable sections should be interpreted as a sequence of dual issue instructions.

3 XMOS executable (XE) file format

IN THIS CHAPTER

The XMOS executable (XE) binary file format holds executable programs compiled to run on XMOS devices. The format supports distinct programs for each xCORE tile in a multi-tile or multi-chip design, and allows multiple loads and runs on each tile.

In addition to the program itself, an XE file contains a description of the system it is intended to run on. This description takes the form of either an XML system configuration description or a 64-bit per-node system identifier.

3.1 Binary format

The following sections explain the common elements of the binary format. All data is encoded as little endian.

3.1.1 XE header

An XE file must start with an XE header. It has the following format:

Figure 13: XE header

Byte offset	Length (bytes)	Description
0x0	4	The string `XMOS` encoded in ASCII.
0x4	1	Major version number (2).
0x5	1	Minor version number (0).
0x6	2	Reserved. Must be set to zero.

3.1.2 Sectors

The XE header is followed by a list of sectors. The end of the sector list must be marked using a sector with a sector type of 0x5555. Each sector consists of a sector header, optionally followed by a variable-length sector contents block containing sector data. Padding is added after the sector data to make the sector contents block a whole number of 32-bit words.

The sector CRC is calculated on the byte stream from the start of the sector header to the byte before the sector CRC. The polynomial used is 0x04C11DB7 (IEEE 802.3); the CRC register is initialized with 0xFFFFFFFF and residue is inverted to produce the CRC.

Byte offset	Length (bytes)	Description
0x0	2	Sector type.
0x2	2	Reserved. Must be set to zero.
0x4	8	Size in bytes of the sector contents block. Set to zero if this sector has no sector contents block.

Figure 14: Sector header

Byte offset	Length (bytes)	Description
0x0	1	Size in bytes of the padding after the sector data.
0x1	3	Reserved. Must be set to zero.
0x4	n	Sector data.
0x4+n	p	Padding bytes to align to the next 32-bit word.
0x4+n+p	4	Sector CRC.

Figure 15: Sector contents block

The following sector types are defined:

Value	Name	Description
0x1	Binary	Load binary image.
0x2	ELF	Load ELF image.
0x3	SysConfig	System description XML.
0x4	NodeDescriptor	Node description.
0x5	Goto	Start execution.
0x6	Call	Start execution and wait for return.
0x8	XN	XN description.
0x5555	Last sector	Marks the end of the file.
0xFFFF	Skip	Skip this sector.

Figure 16: Sector types

The meaning of the sector data depends on the sector type. The following sections provide further details of the format of the sector data for each sector type.

3.1.2.1 SysConfig sectors

The SysConfig sector contains a full XML description of the system, including number of nodes, xCORE tiles and link/interconnect configuration. This information is provided by XMOS to describe its chip products. The format of the SysConfig sector is currently undocumented.

3.1.2.2 Node descriptor sectors

The NodeDescriptor sector describes an individual node, allowing the toolchain to validate an executable file matches the target device. There may be 0 or more NodeDescriptor sectors.

Figure 17: NodeDescriptor sector

Data byte offset	Length (bytes)	Description
0x0	2	Index of the node in the JTAG scan chain.
0x2	2	Reserved.
0x4	4	Device JTAG ID.
0x8	4	Device JTAG user ID.

3.1.2.3 XN sector

The XN sector contains a XN description of the system, as described in the xTIMEcomposer User Guide.[6]

3.1.2.4 Binary/ELF sectors

Binary or ELF sectors instruct the loader to load a program image on the specified xCORE tile. Binary/ELF sectors are formatted as shown in the following table:

Figure 18: Binary/ELF sector

Data byte offset	Length (bytes)	Description
0x0	2	Index of the node in the JTAG scan chain.
0x2	2	xCORE tile number.
0x4	8	Load address of the binary image data. For ELF sectors this field should be set to 0.
0xC	n	Image data.

When a binary sector is loaded the data field is copied into memory starting at the specified load address. When a ELF sector is loaded the loadable segments of ELF image contained in the data field are loaded to the addresses specified in the ELF image.

3.1.2.5 Goto/call sectors

Goto and call sectors instruct the loader to execute code on the specified xCORE tile. If the last image loaded onto the tile was a ELF image execution starts at address of the `_start` symbol, otherwise execution starts at address specified as a field in the sector.

[6] The XMOS xTIMEcomposer user guide. http://www.xmos.com/published/xtools_en.

When processing a call sector the loader should wait for the code to indicate successful termination via a done or exit system call before processing the next sector.

Data byte offset	Length (bytes)	Description
0x0	2	Index of the node in the JTAG scan chain.
0x2	2	xCORE tile number.
0x4	8	Specifies the address to jump to if the last image loaded onto the tile was a binary image. This field should be set to 0 if the last image loaded was an ELF image.

Figure 19: Goto/Call sector

3.1.2.6 Last sector

The last sector type is used to indicate the end of the sector list. A sector of this type should have no sector contents block.

3.1.2.7 Skip sector

A loader must ignore any skip sectors that appear in the sector list. Changing the type of an existing sector to the skip sector type allows removal of sectors without effecting the layout of the XE file.

3.2 Booting an XE File

To boot an XE file the sectors within the file must be processed in sequential order. This allows a loader to load and execute sectors to initialize the system in an order defined by the toolchain, using as many boot stages as required. If an image is loaded onto an xCORE tile there must be exactly one Goto sector. This sector must appear after all Call, Binary and ELF sectors for that tile.

A loader may choose to delay processing of Call sectors until a set of Call sectors have been accumulated for all xCORE tiles on the target device. This allows the loader to reduce boot time by executing as much code as possible in parallel.

Call sectors should only be used to run code that configures interconnect registers. A loader can choose to ignore call sectors so long as it configures the interconnect registers for the target described in the XN sector.

The example in Figure 20 shows a typical layout for an XE file containing a program compiled to run on a XS1-G4 device.

Sector type	Node	Tile	Description
SysConfig			XML System description, ignored by the loader.
XN			XN description, ignored by the loader.
ELF	0	3	Load ELF image onto node 0 tile 3.
Call	0	3	Execute program on node 0 tile 3 and wait for successful termination.
ELF	0	2	Load ELF image onto node 0 tile 2.
Call	0	2	Execute program on node 0 tile 2 and wait for successful termination.
ELF	0	1	Load ELF image onto node 0 tile 1.
Call	0	1	Execute program on node 0 tile 1 and wait for successful termination.
ELF	0	0	Load ELF image onto node 0 tile 0.
Call	0	0	Execute program on node 0 tile 0 and wait for successful termination.
ELF	0	3	Load ELF image onto node 0 tile 3.
Goto	0	3	Execute program on node 0 tile 3.
ELF	0	2	Load ELF image onto node 0 tile 2.
Goto	0	2	Execute program on node 0 tile 2.
ELF	0	1	Load ELF image onto node 0 tile 1.
Goto	0	1	Execute program on node 0 tile 1.
ELF	0	0	Load ELF image onto node 0 tile 0.
Goto	0	0	Execute program on node 0 tile 0.
Last sector			Last sector marker.

Figure 20: Example XE file

4 XMOS system call interface

IN THIS CHAPTER

The xTIMEcomposer tools provide a library containing a set of Unix-like system calls intended to be mapped to operations on an attached host machine. These system calls ease debugging of applications by providing support for file I/O and reporting of program termination.

4.1 System calls

The following system calls are provided:

Function	**_exit**
Description	Indicates the termination of the program with the given exit status. Doesn't return.
Type	`void _exit(int status)`

Function	**_done**
Description	Indicates the successful termination of threads on the current core. Doesn't return. The program is considered to have successfully terminated when `_done` is called on every core on which code is running.
Type	`void _done(void)`

<table>
<tr><td>Function</td><td>_open</td></tr>
<tr><td>Description</td><td>Used to convert a pathname into a file descriptor. The <tt>flags</tt> argument is the the bitwise-or of a subset of the possible open flags. The open flags are given below. One of <tt>O_RDONLY</tt>, <tt>O_WRONLY</tt> or <tt>O_RDWR</tt> must be specified.

If <tt>O_CREAT</tt> is specified the <tt>mode</tt> argument determines of permissions of the file that is created. The <tt>mode</tt> argument is the bitwise-or of a subset of the possible mode flags. The mode flags are given below.

<table>
<tr><td><tt>O_RDONLY</tt></td><td>0x0001</td><td>Read only.</td></tr>
<tr><td><tt>O_WRONLY</tt></td><td>0x0002</td><td>Write only.</td></tr>
<tr><td><tt>O_RDWR</tt></td><td>0x0004</td><td>Read/write enable.</td></tr>
<tr><td><tt>O_CREAT</tt></td><td>0x0100</td><td>Create and open file.</td></tr>
<tr><td><tt>O_TRUNC</tt></td><td>0x0200</td><td>Truncate file to zero bytes.</td></tr>
<tr><td><tt>O_EXCL</tt></td><td>0x0400</td><td>Exclusive open.</td></tr>
<tr><td><tt>O_APPEND</tt></td><td>0x0800</td><td>Open in append mode. Data is written to the end of the file.</td></tr>
<tr><td><tt>O_BINARY</tt></td><td>0x8000</td><td>Open in binary mode.</td></tr>
</table>

<table>
<tr><th>Name</th><th>Value</th><th>Description</th></tr>
<tr><td><tt>S_IREAD</tt></td><td>0400</td><td>Read permission.</td></tr>
<tr><td><tt>S_IWRITE</tt></td><td>0200</td><td>Write permission.</td></tr>
<tr><td><tt>S_IEXEC</tt></td><td>0100</td><td>Execute permission.</td></tr>
</table>

<tt>_open</tt> returns the new file descriptor, or -1 on error (in which case <tt>errno</tt> is set appropriately).</td></tr>
<tr><td>Type</td><td><tt>void _open(const char *pathname, int flags, mode_t mode)</tt></td></tr>
</table>

Function	_close
Description	Closes the file descriptor `fd`. Returns 0 on success, or -1 on error (in which case `errno` is set appropriately).
Type	`void _close(int fd)`

Function	_read
Description	Attempts to read `count` bytes from the file descriptor `fd` to the buffer starting at `buf`. Returns the number of bytes read, or -1 on error (in which case `errno` is set appropriately).
Type	`void _read(int fd, void *buf, size_t count)`

Function	_write
Description	Attempts to write `count` bytes from the buffer starting at `buf` to the file descriptor `fd`. Returns the number of bytes written, or -1 on error (in which case `errno` is set appropriately).
Type	`void _write(int fd, void *buf, size_t count)`

Function	_lseek
Description	Repositions the offset of the file descriptor `fildes` based on the value of `offset` and `whence`. The allowable values of `whence` are given below. **Name** / **Value** / **Description** `SEEK_CUR` / 1 / The offset is set to its current position plus `offset` bytes. `SEEK_END` / 2 / The offset is set to the end of the file plus `offset` bytes. `SEEK_SET` / 0 / The offset is set to `offset` bytes. `_lseek` returns the new offset in bytes from the beginning of the file, or -1 on error (in which case `errno` is set appropriately).
Type	`void _lseek(int fildes, off_t offset, int whence)`

Function	_time
Description	Returns the time since 00:00:00 UTC, January 1, 1970 measured in seconds, or -1 on error. If `t` is non-NULL the return value is also stored in `*t`.
Type	`time_t _time(time_t *t)`

Function	_system
Description	Behaves as the `system` function defined in the C standard.[7]
Type	`void _system(const char *string)`

Function	_remove
Description	Behaves as the `remove` function defined in the C standard. [6]
Type	`void _remove(const char *pathname)`

Function	_rename
Description	Behaves as the `rename` function defined in the C standard. [6]
Type	`void _rename(const char *oldpath, const char *newpath)`

Function	_is_simulation
Description	Returns 1 if the program is being executed in a simulator and 0 otherwise.
Type	`int _is_simulation(void)`

[7] ISO/IEC 9899:1999: Programming Languages - C. Wiley, West Sussex, England, December 1999.

<table>
<tr><td>Function</td><td>_plugins</td></tr>
<tr><td>Description</td><td>Used to control simulator plugins. The <code>type</code> argument determines the action taken. Values less than 65536 are reserved. Values greater than or equal to 65536 can be used to implement user defined communication with custom simulator plugins. The standard values are given below.
<table>
<tr><th>Name</th><th>Value</th><th>Description</th></tr>
<tr><td><code>NOTIFY_PLUGINS_START_TRACE</code></td><td>0</td><td>Instructs plugins to start tracing. Takes no arguments.</td></tr>
<tr><td><code>NOTIFY_PLUGINS_STOP_TRACE</code></td><td>1</td><td>Instructs plugins to stop tracing. Takes no arguments.</td></tr>
</table></td></tr>
<tr><td>Type</td><td><code>void _plugins(int type, unsigned arg1, unsigned arg2)</code></td></tr>
</table>

4.2 System call interface

System calls are performed by calling the `_DoSyscall` function. This enables all system calls to be caught using a single instruction breakpoint. *r0* identifies the system call that should be invoked. The permitted values are given in Figure 21.

Figure 21: System call types

Name	Value
OSCALL_EXIT	0
OSCALL_OPEN	3
OSCALL_CLOSE	4
OSCALL_READ	5
OSCALL_WRITE	6
OSCALL_DONE	7
OSCALL_LSEEK	8
OSCALL_RENAME	9
OSCALL_TIME	10
OSCALL_REMOVE	11
OSCALL_SYSTEM	12
OSCALL_IS_SIMULATION	99
OSCALL_PLUGINS	100

System call arguments are passed to `_DoSyscall` in registers *r1*, *r2* and *r3*. The return value of the system call and the value of `errno` is encoded in the value of *r0* after `_DoSyscall` returns.

If the system call can cause `errno` to be set and the value of `r0` after `_DoSyscall` returns is in the range -128 to -1 inclusive `errno` is set to the negation of *r0* and -1 is the return value. If value of

r0 is outside this range or if the system call does not set errno then the value of *r0* is used as the return value. The allowable values of `errno` are given in Figure 22.

Figure 22: `errno` values

Name	Value	Description
EPERM	1	Operation not permitted.
ENOENT	2	No such file or directory.
EINTR	4	Interrupted system call.
EIO	5	I/O error.
ENXIO	6	No such device or address.
EBADF	9	Bad file number.
EAGAIN	11	Try again.
ENOMEM	12	Out of memory.
EACCES	13	Permission denied.
EFAULT	14	Bad address.
EBUSY	16	Device or resource busy.
EEXIST	17	File exists.
EXDEV	18	Cross-device link.
ENODEV	19	No such device.
ENOTDIR	20	Not a directory.
EISDIR	21	Is a directory.
EINVAL	22	Invalid argument.
ENFILE	23	File table overflow.
EMFILE	24	Too many open files.
ETXTBSY	26	Text file busy.
EFBIG	27	File too large.
ENOSPC	28	No space left on device.
ESPIPE	29	Illegal seek.
EROFS	30	Read-only file system.
EMLINK	31	Too many links.
EPIPE	32	Broken pipe.
EDOM	33	Math argument out of domain of function.
ERANGE	34	Math result not representable.
ENAMETOOLONG	36	File name too long.
ENOSYS	38	Function not implemented.
ENOTEMPTY	39	Directory not empty.
ELOOP	40	Too many symbolic links encountered.
EOVERFLOW	75	Value too large for defined data type.
EILSEQ	84	Illegal byte sequence.

4.3 Exceptions

In the event of an exception the default exception handler branches to the `_DoException` function. This enables all exceptions to be caught using a single instruction breakpoint. All registers apart from the *pc* have the values they had immediately after the exception was taken.