

## 19 XC, C and C++

The XMOS toolchain supports the following languages:

<b>XC</b>	XMOS XC Version 9.9 [2]
<b>C89</b>	ISO/IEC 9899:1990 standard [3]
<b>C99</b>	ISO/IEC 9899:1999 standard [4, 5] (partial support)
<b>C++</b>	ISO C++ standard (1998) [6]
<b>GNUC</b>	GNU C/C++ extensions [7]

The toolchain targets XMOS XS1-based systems [8], implementing the XMOS XS1 Application Binary Interface [9]. Target platforms are described using XN (see §21).

### 19.1 Pragma Directives

The XMOS toolchain supports the following pragmas:

`#pragma unsafe arrays` (XC only)

This pragma disables the generation of run-time safety checks that prevent indexing an invalid array element within the scope of the next `do`, `while` or `for` statement in the current function; outside of a function the pragma applies to the next function definition.

`#pragma loop unroll (n)` (XC only)

This pragma controls the number of times the next `do`, `while` or `for` loop in the current function is unrolled. *n* specifies the number of iterations to unroll, and unrolling is performed only at optimisation level 01 and higher. Omitting the (*n*) parameter will cause the compiler to try and fully unroll the loop. Outside of a function the pragma is ignored. The compiler produces a warning if unable to perform the unrolling.

- `#pragma stackfunction n`  
This pragma allocates *n* words (`ints`) of stack space for the next function declaration in the current translation unit.
- `#pragma stackcalls n (XC only)`  
This pragma allocates *n* words (`ints`) of stack space for any function called in the next statement. If the next statement does not contain a function call then the pragma is ignored; the next statement may appear in another function.
- `#pragma ordered (XC only)`  
This pragma controls the compilation of the next `select` statement. This select statement will be compiled in a way such that if multiple events are ready when the select starts, cases earlier in the select statement are selected in preference to ones later on.
- `#pragma fallthrough (XC only)`  
This pragma indicates that the following switch case is expected to fall-through to the next switch case without a `break` or `return` statement. This will suppress any warnings/errors from the compiler due to the fallthrough.
- `#pragma xta ...`  
Pragmas prefixed with `xta` indicate information relating to timing analysis. Refere to XTA documentation for details.

## 19.2 Inline Assembly

The `asm` statement can be used to embed code written in assembly inside a C or XC function. For example the add instruction can be written as follows:

```
asm("add %0, %1, %2" : "=r"(result) : "r"(a), "r"(b));
```

Colons separate the assembler template, the output operands and the input operands. Commas separate operands within a group. Each operand is described by an operand constraint string followed by an expression in parentheses. The “`r`” in the operand constraint string indicates that the operand must be located in a register. The “`=`” in the operand constraint string indicates that the operand is written.

Each output operands expression must be an lvalue and must have “`=`” in its constraint.

The location of an operand may be referred to in the assembler template using an escape sequence of the form `%num` where `num` is the operand number. To produce a literal “`%`” you must write “`%%`”.

If code overwrites specific registers this can be described by using a third colon after the input operands, followed by the names of the clobbered registers as a comma separated list of strings. For example:

```
asm("getid r11\n\tmov %0, r11"
    : "=r"(result)
```

```
    : /* no inputs */  
    : "r11");
```

The compiler ensures none of input or output operands are placed in the clobbered registers. None of the output operands will be placed in the same location. The compiler assumes all inputs are consumed before the outputs are produced and so an output operand may be allocated to the same register as an unrelated input operand.

## 19.3 XC Implementation-Defined Behaviour

A conforming XC implementation is required to document its choice of behaviour for all parts of the language specification that are designated *implementation-defined*. In the following section, all choices that depend on an externally determined application binary interface are listed as “determined by ABI,” and are documented in the XC programming guide [10, Appendix C].

- **The value of a multi-character constant (A.1.5.2).**  
The value of a multi-character constant is the same as the value of its first character; all other characters are ignored.
- **Whether identical string literals are distinct (A.1.6).**  
Identical string literals are not distinct; they are implemented in a single location in memory.
- **The extent to which suggestions made by using the `inline` function specifier are effective (A.3.1).**  
The `inline` function specifier is taken as a hint to inline the function. The compiler will try and inline the function at all optimization levels above `-O0`. An `inline` specifier without a `extern` or `static` specifier is interpreted as an inline definition with linkage defined as in C99.
- **The available range of values that may be stored into a `char` and whether the value is signed (A.3.2).**  
The size of `char` is 8 bits. Whether values stored in a `char` variable are signed or not is determined by the ABI.
- **The number of pins interfaced to a port and used for communicating with the environment; and the value of a port or clock not declared `extern` and not explicitly initialised (A.3.2, A.7.7).**  
The number of pins connected to a port for communicating with the environment is defined either by the explicit initialiser for its declarator, if present, or a value passed to the mapper (see §8.7.7).
- **The notional transfer type of a port, the notional counter type of a port, and the notional counter type of a timer (A.3.2).**

The notional types are determined by the ABI.

- **The value of an integer converted to a signed type such that its value cannot be represented in the new type (A.5.2).**

When any integer is converted to a signed type and its value cannot be represented in the new type, its value is truncated to the width of the new type and sign extended.

- **The handling of overflow, divide check, and other exceptions in expression evaluation (A.6).**

Divide by zero and modulo zero both trap. For division and modulo, if the result of the operation (after application of the usual arithmetic conversions) is `(signed int)-0x80000000` and the value of the divisor is `-1` then the operation traps. These traps are not considered side effects.

- **The notion of alignment (A.6.3.4).**

An alignment of  $2^n$  guarantees that the least significant  $n$  bits of the address in memory are 0. The specific alignment of the types is determined by the ABI.

- **The value and the type of the result of `sizeof` (A.6.4.6).**

The value of the result of the `sizeof` operator is determined by the ABI. The type of the result is `unsigned int`.

- **Attempted run-time division by zero (A.6.6).**

Attempted run-time division by zero results in a trap.

- **The extent to which suggestions made by using the `register storage class specifier` are effective (A.7.6.3).**

The `register storage class specifier` will cause the register allocator to try and place the variable in a register within the function. However, the allocator is not guaranteed to place it in a register.

- **The supported predicate functions for input operations (A.8.3).**

The set of supported predicate functions is documented in the XC programming guide [10, Appendix C].

- **The extent to which the underlying communication protocols are optimised for transaction communications (A.8.9).**

The communication protocols are determined by the ABI.

- **Whether a transaction is invalidated if a communication occurs such that the number of bytes output is unequal to the number of byte input, and the value communicated (A.11).**

This is determined by the ABI.

— **The behaviour of an invalid operation (A.13).**

Except as described below, all invalid operations are either reported as compilation errors or cause a trap at run-time.

- The behaviour of an invalid master transaction statement is undefined; an invalid slave transaction always traps.
- The `unsafe` pragma (see §19.1) can be used to disable specific safety checks, resulting in undefined behaviour for invalid operations.

## 19.4 XS1 Target-Dependent Behaviour

This section describes behaviour of the compiler which depends on the target architecture.

- When compiling for the XS1-G architecture the compiler disallows selecting on a channel input of less than a word-length in a streaming channel. The command line option `-fsubword-select` relaxes this restriction, but this can lead to cases with these functions not being taken even if data is available on the channel.
- When compiling for the XS1-G architecture the `inuchar_byref`, `inct_byref` and `testct` functions may not be used in a select. The command line option `-fsubword-select` relaxes this restriction, but this can lead to cases with these functions not being taken even if data is available on the channel.
- G2 and G4 devices can only be placed in degree  $n$  hypercubes where  $n \leq 4$  — a single node, two nodes, a ring of four, a cube (8 nodes) or a canonical hypercube (16 nodes).
- L1 devices can only be arranged in hypercubes and lines of up to 16.

## 19.5 C Implementation-Defined Behaviour

A conforming C99 implementation is required to document its choice of behaviour for all parts of the language specification that are designated *implementation-defined*. The XMOS toolchain implementation-defined behaviour matches that of GCC 4.2.1 [1] except for the choices listed below.

The following section headings refer to sections in the C99 specification [5] and all choices that depend on an externally determined application binary interface are listed as “determined by ABI.” Only the supported C99 features are documented.

### 19.5.1 Environment

— **The name and type of the function called at program startup in a free-standing environment (5.1.2.1).**

A hosted environment is provided.

- **An alternative manner in which the `main` function may be defined (5.1.2.2.1).**  
There is no alternative manner in which `main` may be defined.
- **The values given to the strings pointed to by `argv` argument to `main` (5.1.2.2.1).**  
The value of `argc` is equal to zero. `argv[0]` is a null pointer. There are no other array members.
- **What constitutes an interactive device (5.1.2.3).**  
All streams are refer to interactive devices.
- **Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1).**  
No other signal values correspond to a computational exception.
- **Signal values for which is equivalent of `signal(sig, SIG_IGN)`; is executed at program startup (7.14.1.1).**  
At program startup the equivalent of `signal(sig, SIG_DFL)`; is executed for all signals.
- **The set of environment names and the method for altering the environment list used by the `getenv` function (7.20.4.5).**  
The set of environment names is empty. There is no method for altering the environment list used by the `getenv` function.
- **The manner of execution of the string by the `system` function used by the `getenv` function.(7.20.4.6).**  
This is determined by the execution environment.

### 19.5.2 Identifiers

- **The number of significant initial characters in an identifier (5.2.4.1, 6.4.1).**  
All characters in identifiers (with or without external linkage) are significant.

### 19.5.3 Characters

- **The value of the members of the execution character set (5.2.1).**  
This is determined by the ASCII character set.
- **The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).**

This is determined by the ASCII character set.

- **The value of a `char` object into which has been stored any character other than a member of the basic execution set (6.2.5).**

The value of any character other than a member of the basic execution set is determined by the ASCII character set.

- **The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).**

The source character set is required to be the ASCII character set. Each character in the source character set is mapped to the same character in the execution character set.

- **The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).**

The value of an integer character constant containing more than one character is equal to the value of the last character in the character constant. The value of an integer character constant containing a character or escape sequence that does not map to a single-byte execution character is equal to the value reduced modulo  $2^N$  to be within range of the `char` type, where  $N$  is the number of bits in a `char`.

- **The value of a wide character constant containing more than one multi-byte character, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).**

Wide character constants must not contain multibyte characters.

- **The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).**

Wide character constants must not contain multibyte characters.

- **The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (6.4.5).**

String literals must not contain multibyte characters. If an escape sequence not represented in the execution character set is used in a string literal then the value of the corresponding character in the string is the same as the value that would be given to an integer character constant consisting of that escape sequence.

#### 19.5.4 Floating point

- **The accuracy of the floating-point operations and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results**

(5.2.4.2.2).

This is intentionally left undocumented.

- **Additional floating-point exceptions, rounding modes, environments, and classifications, and their macros names (7.6, 7.12).**

No additional floating-point exceptions, rounding modes, environments or classifications are defined.

### 19.5.5 Hints

- **The extent to which suggestions made by using the `register storage-class specifier` are effective (6.7.1).**

The `register` specifier is ignored except when used as part of the register variable extension.

### 19.5.6 Preprocessing directives

- **The behavior on each recognized non-STDC `#pragma` directive (6.10.6).**

This is documented in §19.1.

### 19.5.7 Library functions

- **Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1).**

A hosted environment is provided.

- **The format of the diagnostic printed by the `assert` macro (7.2.1.1).**

The `assert` macro uses the format "Assertion failed: *expression*, file *filename*, line *line number*, function: *function*." where *expression* is the text of the argument, *filename* is the value of `__FILE__`, *line number* is the value of `__LINE__` and *function* is the name of the current function. If the name of the current function cannot be determined then this part of the message will be omitted.

- **The representation of the floating-point status flags stored by the `fegetexceptflag` function (7.6.2.2).**

The function `fegetexceptflag` is not supported.

- **Whether the `feraiseexcept` function raises the "inexact" floating-point exception in addition to the "overflow" and "underflow" floating-point exception (7.6.2.3).**

The function `feraiseexcept` is not supported.

- **Strings other than "C" and "" that may be passed as the second argument to the `setlocale` function (7.11.1.1).**  
No other strings may be passed as the second argument to the `setlocale` function.
- **The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 or greater than 2 (7.12).**  
No other values of the `FLT_EVAL_METHOD` macro are supported.
- **Domain errors for the mathematics functions, other than those required by this International Standard (7.12.1).**  
This is intentionally left undocumented.
- **The values returned by the mathematics functions on domain errors (7.12.1).**  
This is intentionally left undocumented.
- **The values returned by the mathematics functions on underflow range errors, whether `errno` is set to the value of the macro `ERANGE` when the integer expression `math_errhandling & MATH_ERRNO` is nonzero, and whether the "underflow" floating-point exception is raised when the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero (7.12.1).**  
This is intentionally left undocumented.
- **Whether a domain error occurs or zero is returned when an `fmod` function has a second argument of zero (7.12.10.1).**  
A domain error occurs when an `fmod` function has a second argument of zero.
- **The base-2 logarithm of the modulus used by the `remquo` functions in reducing the quotient (7.12.10.3).**  
The quotient is reduced modulo  $2^7$ .
- **Whether the equivalent of `signal(sig, SIG_DFL)`; is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1).**  
The equivalent of `signal(sig, SIG_DFL)`; is executed prior to the call of a signal handler.
- **The null pointer constant to which the macro `NULL` expands (7.17).**  
`NULL` is defined as `((void *)0)`.
- **Whether the last line of a text stream requires a terminating new-line character (7.19.2).**

This is determined by the execution environment.

- **Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.19.2).**

This is determined by the execution environment.

- **The number of null characters that may be appended to data written to a binary stream (7.19.2).**

This is determined by the execution environment.

- **Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of a file (7.19.3).**

This is determined by the execution environment.

- **Whether a write on a text stream causes the associated file to be truncated beyond that point (7.19.3).**

This is determined by the execution environment.

- **The characteristics of file buffering (7.19.3).**

A buffered output stream saves characters until the buffer is full and then writes the characters as a block. A line buffered output stream saves characters until the line is complete or the buffer is full and then writes the characters as a block. An unbuffered output stream writes characters to the destination file immediately.

- **Whether a zero-length file actually exists (7.19.3).**

This is determined by the execution environment.

- **The rules for composing valid file names (7.19.3).**

This is determined by the execution environment.

- **Whether the same file can be simultaneously opened multiple times (7.19.3).**

This is determined by the execution environment.

- **The nature and choice of encodings used for multibyte characters in files (7.19.3).**

The execution character set must not contain multibyte characters.

- **The effect of the `remove` function on an open file (7.19.4.1).**

This is determined by the execution environment.

- **The effect if a file with the new name exists prior to a call to the `rename` function (7.19.4.1).**

This is determined by the execution environment.

- **Whether an open temporary file is removed upon abnormal program termination (7.19.4.3).**

Temporary files are not removed on abnormal program termination.

- **Which changes of mode are permitted (if any), and under what circumstances (7.19.5.4).**

The file cannot be given a more permissive access mode (for example, a mode of “w” will fail on a read-only file descriptor), but can change status such as append or binary mode. If modification is not possible, failure occurs.

- **The style used to print an infinity or NaN, and the meaning of any n-char or n-wchar sequence printed for a NaN (7.19.6.1, 7.24.2.1).**

A `double` argument representing infinity is converted in the style `[-]inf`. A `double` argument representing a `NaN` is converted in the style `nan`.

- **The output for %p conversion in the fprintf or fwprintf function (7.19.6.1, 7.24.2.1).**

The value of the pointer is converted to unsigned hexadecimal notation in the style `dddd`; the letters `abcdef` are used for the conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The characters `0x` are prepended to the output.

The `fwprintf` function is unsupported.

- **The interpretation of a - character that is neither the first nor the last character, nor the second where a ^ character is the first, in the scanlist for %[ conversion in the fscanf or fwscanf function (7.19.6.2, 7.24.2.1).**

The `-` character is considered to define a range if the character following it is numerically greater than the character before it. Otherwise the `-` character itself is added to the `scanset`.

The `fwscanf` function is unsupported.

- **The set of sequences matched by a %p conversion and the interpretation of the corresponding input item in the fscanf or fwscanf function (7.19.6.2, 7.24.2.2).**

`%p` matches the same format as `%x`. The corresponding input item is converted to a pointer.

The `fwscanf` function is unsupported.

- **The meaning of any n-char or n-wchar sequence in a string representing NaN that is converted by the strtod, strtodf, strtold, wcstod, wcstof or wcstold function (7.20.1.3, 7.24.4.1.1).**

The functions `wcstod`, `wcstof` and `wcstold` are not supported. A `n-char` sequence in a string representing *NaN* is scanned in hexadecimal form. Any characters which are not hexadecimal digits are ignored.

- **Whether or not the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof` or `wcstold` function sets `errno` to `ERANGE` when underflow occurs (7.20.1.3, 7.24.4.1.1).**

The functions `wcstod`, `wcstof` and `wcstold` are not supported. The functions `strtod`, `strtof` and `strtold` do not set `errno` to `ERANGE` when and return 0 underflow occurs.

- **Whether the `calloc`, `malloc`, and `realloc` functions return a null pointer or a pointer to an allocated object when the size requested is zero (7.20.3).**

The functions `calloc`, `malloc` and `realloc` functions all return a pointer to an allocated object when the size requested is zero.

- **Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed when the `abort` or `_Exit` function is called (7.20.4.1, 7.20.4.3, 7.20.4.4).**

When the `abort` function or `_Exit` function is called, temporary files are not removed, buffered files are not flushed and open streams are left open.

- **The termination status returned to the host environment by the `abort`, `exit` or `_Exit` function (7.20.3).**

The function `abort` causes a software exception to be raised. The termination status returned to the host environment by the functions `exit` and `_Exit` is determined by the execution environment.

- **The value returned by the `system` function when its argument is not a null pointer (7.20.4.6).**

This is determined by the execution environment.

- **The range and precision of times representable in `clock_t` and `time_t` (7.23.1).**

The precision of times representable in `time_t` is defined by the execution environment. `time_t` designates an unsigned long. The actual range of times representable by `time_t` is defined by the execution environment.

The macro `CLOCKS_PER_SEC` is defined as 1000. `clock_t` designates an unsigned long.

- **The era for the `clock` function (7.23.2.1).**

The `clock` function always returns the value `(clock_t)(-1)` to indicate that the processor time used is not available.

- **The replacement string for the `%Z` specifier to the `strftime` and `wcsftime` functions in the "C" locale (7.23.3.5, 7.24.5.1).**

The %Z specifier is replaced with the string “GMT”.

### 19.5.8 Locale-Specific Behavior

- **Additional members of the source and execution character sets beyond the basic character set (5.2.1).**

Both the source and execution character sets include all members of the ASCII character set.
- **The presence, meaning, and representation of additional multibyte characters in the execution character set beyond the basic character set (5.2.1.2).**

The execution character set does not contain multibyte characters.
- **The shift states used for the encoding of multibyte characters (5.2.1.2).**

The source and execution character sets does not contain multibyte characters.
- **The direction of writing of successive printing characters (5.2.2).**

Characters are printed from left to right.
- **The decimal-point character (7.1.1).**

The decimal-point character is ‘.’.
- **The set of printing characters (7.4, 7.25.2).**

This is determined by the ASCII character set.
- **The set of control characters (7.4, 7.25.2).**

This is determined by the ASCII character set.
- **The set of characters tested for by the `isalpha`, `isblank`, `islower`, `ispunct`, `isspace`, `isupper`, `iswalpha`, `iswblank`, `iswlower`, `iswpunct`, `iswspace`, or `iswupper` functions (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.2.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11).**

The functions `isblank`, `iswalpha`, `iswblank`, `iswlower`, `iswpunct`, `iswspace` and `iswupper` are not supported.

`islower` tests for the characters ‘a’ to ‘z’. `isupper` tests for the characters ‘A’ to ‘Z’. `isspace` tests for the characters ‘ ’, ‘\f’, ‘\n’, ‘\r’, ‘\t’ and ‘\v’. `isalpha` tests for upper and lower case characters. `ispunct` tests for all printable characters except space and alphanumeric characters.
- **The native environment (7.11.1.1).**

The native environment is the same as the minimal environment for C translation.

- **Additional subject sequences accepted by the numerical conversion functions (7.20.1, 7.24.4.1).**  
No additional subject sequences are accepted by the numerical conversion functions.
- **The collation sequence of the execution character set (7.21.4.3, 7.24.4.2).**  
The comparison carried out by the function `strcoll` is identical to the comparison carried out by the function `strcmp`.
- **The contents of the error message strings set up by the `strerror` function (7.21.4.3, 7.24.4.2).**

Value	String
EPERM	Not owner
ENOENT	No such file or directory
EINTR	Interrupted system call
EIO	I/O error
ENXIO	No such device or address
EBADF	Bad file number
EAGAIN	No more processes
ENOMEM	Not enough space
EACCES	Permission denied
EFAULT	Bad address
EBUSY	Device or resource busy
EEXIST	File exists
EXDEV	Cross-device link
ENODEV	No such device
ENOTDIR	Not a directory
EISDIR	Is a directory
EINVAL	Invalid argument
ENFILE	Too many open files in system
EMFILE	Too many open files
ETXTBSY	Text file busy
EFBIG	File too large
ENOSPC	No space left on device
ESPIPE	Illegal seek
EROFS	Read-only file system
EMLINK	Too many links
EPIPE	Broken pipe
EDOM	Math argument
ERANGE	Result too large
ENAMETOOLONG	File or path name too long
ENOSYS	Function not implemented
ENOTEMPTY	Directory not empty
ELOOP	Too many symbolic links

— **The formats for time and date (7.23.3.5, 7.24.5.1).**

The time and date are formatted according to the replacement strings specified by the standard for the "C" locale, regardless of the current locale.

— **Character mappings that are supported by the `towctrans` function (7.25.1).**

Value	Description
WCT_TOLOWER	Convert to lower case
WCT_TOUPPER	Conver to upper case

- **Character classifications that are supported by the `iswctype` function (7.25.1).**

Value	Description
WCT_ALNUM	Alphanumeric characters
WCT_ALPHA	Alphabetic characters
WCT_BLANK	Blank characters
WCT_CNTRL	Control characters
WCT_DIGIT	Decimal digits
WCT_GRAPH	Characters with a visible representation
WCT_LOWER	Lower case characters
WCT_PRINT	Printable characters
WCT_PUNCT	Punctuation characters
WCT_SPACE	White-space characters
WCT_UPPER	Upper case characters
WCT_XDIGIT	Hexadecimal digits

## 19.6 Implementation support for XC §9.6 & §9.10

If a function  $f$  is defined then the symbol  $f.actnoside$  may be used in assembly code without being defined elsewhere. In this case at final-link time the value of  $f.actnoside$  is computed as follows: the mapper constructs the program call graph; if any function in  $f$ 's call graph,  $x$ , has the symbol  $x.locnoside$  or  $x.actnoside$  defined and set to 0 then its value is 0; otherwise its value is 1. Similarly, the symbol  $f.actnochandec$  may be used without being defined elsewhere. Its value is computed at final-link time in the same way from definitions of  $x.locnochandec$ .