C-to-Hardware Compiler User Manual

GU0122 May 19, 2008

C-to-Hardware Compiler User Manual

Table of Contents

Chapter 1. Introduction

1.1. Manual Purpose and Structure

The purpose of this manual is to provide detailed information on using the C-to-Hardware (CHC) Compiler in Altium Designer.

This manual describes the hardware compiler functionality in detail. All this is intended to help you make good design choices when creating your ultimate FPGA design.

1.1.1. Required Knowledge to use the CHC Compiler

Familiarity with the C programming language is essential. Experience with optimizing your code for a given target processor architecture helps to decide which code fragments would probably benefit most from compilation to hardware. Knowledge about hardware design languages is *not* required.

After compilation, the generated HDL file must be integrated with the rest of the hardware design. Subsequently the resulting design must be instantiated on the FPGA. In Altium Designer this process is fully automated.

1.1.2. Suggested Reading

We suggest to read the following manuals as well. They provide an introduction to the C-to-Hardware Compiler and to the Application Specific Processor which holds compiled hardware functions:

- *TU0130 Getting Started with the C-to-Hardware Compiler*
- *CR0177 WB_ASP Configurable Application Specific Processor*

You can find these documents in the `...\Help` directory of Altium Designer's installation directory. You can either access them from there directly, or locate and launch them from the lower region of the Knowledge Center (From the **Help** menu, select **Knowledge Center**). You can then find the documents by navigating to Embedded Processors and Software Development » Accelerating Processors with C-to-Hardware.

1.2. Introduction to C-to-Hardware Compilation

1.2.1. Todays FPGAs

Today's low cost FPGA devices contain over one million logic gates, tens or hundreds of dedicated functional units (such as multipliers) mega-bits of memory, and host soft processor cores that occupy less than a few percent of the available logic gates. Electronic systems that employ the latest FPGA technology can provide extremely high computational throughput, ride Moore's law curve, and adapt to change.

To design and program systems that exploit the power offered by these devices is complex and challenging. Three design paradigms are common for FPGA based system development: HDL based design, platform based design, and algorithmic design. Each paradigm has its own strengths and weaknesses with respect to quality, time, and costs of result.

1.2.2. CHC Programming Paradigm

The C-to-Hardware Compiler is an algorithmic design and implementation tool which reduces the cost and lead time of the design cycle while maintaining quality of results. The C-to-Hardware Compiler accepts standard untimed ISO-C source code as input. The C-to-Hardware compiler can either translate a C source file to hardware or translate parts of the C source to hardware functions and the remaining parts to an instruction sequence for a micro controller. In this latter case, the C-to-Hardware Compiler is used in combination with one of Altium Designer's traditional embedded compilers to build a system with an embedded processor core(s) that off-loads certain functions to hardware. With the function qualifier `__rtl` in front of a function definition you tell the compiler to translate the function into a hardware function. This hardware function can either be a callable 'function' (*Application Specific Processor* or: *ASP*) that interacts with the embedded software, or it can be an independent module of FPGA logic (a *C code symbol*) with input and output ports to connect it as a part into a larger FPGA design.

In order to do hardware-software co-design while not modifying any source code, these qualifiers can be put into a qualifier file. In Altium Designer, you can select the functions for hardware compilation from your embedded C source, or you can add a C code symbol to the FPGA sheet and program it. Altium Designer then automatically creates the qualifier file using the information and settings from the FPGA sheet. The compiler will produce a synthesizable Register Transfer Level (RTL) file. An RTL file describes the electronic circuit. Synthesis tools translate this RTL to an electronic circuit that implements the function.

1.2.3. Benefits of the CHC Compiler

The benefits of C-to-Hardware translation are quite diverse and manifest themselves at various levels in the design process.

C-to-Hardware translation technology unites the disparate domains of system, software and hardware engineering. System designs expressed in C can be instantly translated into Register Transfer Level (RTL) descriptions enabling an early exploration of the design space to find an optimal software/hardware partitioning. Design decisions at this level usually have significant impact on product costs and performance.

Near the end of the development cycle, software engineers have few options to improve system performance. Rewriting code in assembly or implementing more efficient algorithms is time consuming and costly. C-to-Hardware compilation provides the software engineer with an automated method to off-load time critical software functions to hardware without having to redesign/rewrite any source code. Off-loading performance critical functions to hardware may also enable you to use a processor core that costs less and/or runs at a lower frequency, potentially reducing power consumption and electromagnetic interference.

Replacing a traditional HDL based hardware design flow with a C-based design flow can result in a steep reduction in development costs and/or time-to-market. RTL produced by the C compiler is 'correct by construction'; time consuming simulation sessions to track down errors in handwritten RTL becomes a thing of the past. Whether these benefits materialize, depends on the type of hardware that is designed.

Hardware components implementing computational complex algorithms can be efficiently described in C, and the quality of the compiler generated RTL approaches the quality of handwritten code. For demanding applications where hand optimized RTL is required, C-to-Hardware translation is still an efficient tool to analyze the performance of different micro-architectures before committing to one and implementing it in a hardware design language.

C-based design is not always a replacement for HDL-based design. Hardware components that are modeled in the structural and/or geometric domain are not easily described in, nor efficiently inferred from, the C language.

Nowadays, computational intensive designs are still implemented using DSP processor cores, whereas an FPGA would outperform the DSP in terms of throughput and costs. Microcontrollers typically offer one execution pipeline, and DSPs up to eight. Modern FPGA's on the other hand, offer virtually unlimited computational resources that can execute in parallel. Since the instruction fetch and decode steps become superfluous and do not appear when an algorithm is implemented in hardware, all available memory bandwidth is available for data access. Such dedicated hardware created by the CHC compiler may outperform traditional microcontrollers and DSPs by orders of magnitude. However, the FPGA design flow is often considered as immature, complex and too risky to use. The CHC compiler solves this problem by offering a traditional embedded/DSP C-based design flow to develop the FPGA circuitry.

Replacing the hardware oriented FPGA design flow with a software design flow broadens the community that could design ASIC-like technology significantly. It particularly lowers the entry barriers for small and mid-sized companies to use advanced FPGA technology.

1.2.4. What can you expect from CHC compiler?

The CHC compiler is designed to be used together with Altium Designer's traditional embedded compilers to create systems that contain both hardware and software. In addition, it can be used to create FPGA logic from a C source. Such module of FPGA logic (a C code symbol) will be part of the larger FPGA design and does not directly interact with the embedded software.

In essence the CHC compiler is a high-optimizing general purpose C-to-gates compiler, extended with facilities to easily interface the generated logic with a processor core.

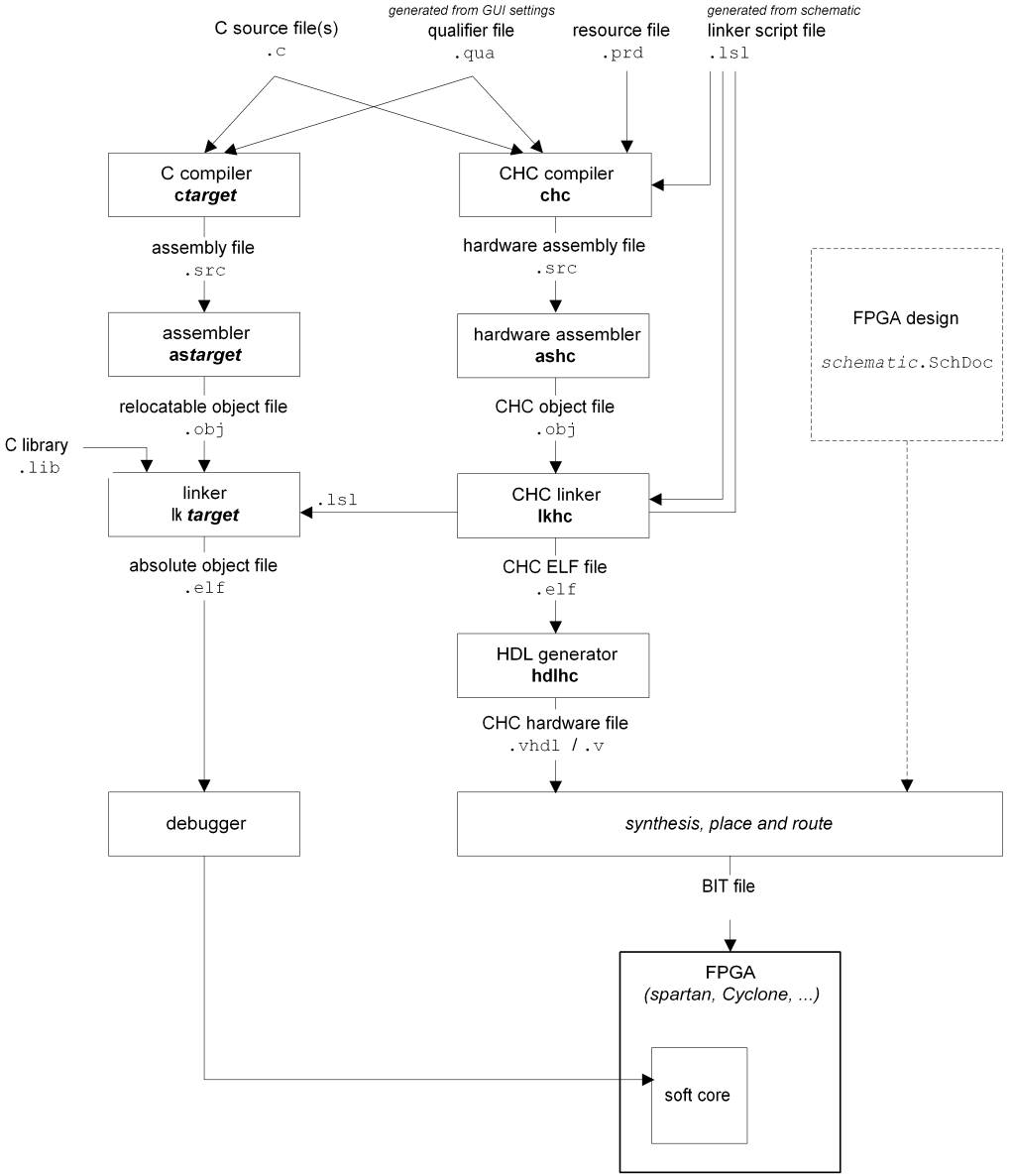
Virtually all C programs can be converted to an electronic circuit by the CHC compiler. However, the characteristics of the program ultimately determine whether the CHC compiler can create an efficient hardware component or whether it is better to execute the program on a processor core. The CHC compiler can only create a small and fast electronic circuit if the C source code is *parallelizable*, in such cases the hardware executes many operations in parallel whereas a processor core would fetch and execute instructions sequentially.

Graphics, signal processing , filter and encryption algorithms typically translate very well to hardware and performance improves by orders of magnitude. For these types of algorithms FPGA implementations outperform high-end DSP and RISC processor cores.

1.3. Toolset Overview

The figure below shows the CHC toolset (right) and its relation to a regular embedded toolset (left). This is the flow to create an Application Specific Processor (a combination of software and hardware functions). For C code symbols (a hardware 'function' as a customized FPGA component specified in C language), only the right part of the figure applies.

Figure 1.1. CHC Toolset Flow



1.3.1. Compiling to Hardware

The C source files are submitted to the compiler of the CHC toolset and -if necessary- to the compiler of a regular embedded toolset. The embedded toolset compiles the C sources to software, whereas the hardware compiler compiles certain functions and data objects to hardware.

For this, both compilers need to know which functions and/or data objects should be translated to hardware and which should be translated to software. The compilers have two ways of knowing this:

- The compiler reads special qualifiers that you can use in the C source files to mark functions and/or data objects for hardware compilation, or,
- the compiler reads a qualifier file which is a list of functions and their associated qualifiers. This file is generated when you use the dialogs in Altium Designer to mark which functions and data objects should be compiled to hardware.

The CHC compiler also reads a *linker script language file* (LSL file). An LSL file describes the target architecture in terms of memory spaces and buses, information a linker/locator uses to locate software and data sections. An embedded compiler does not need such an LSL file, but the CHC compiler does!

Because memory access is the weakest link in FPGA performance, the CHC compiler uses the information in the LSL file to construct a high-performance memory system. To exploit the bandwidth, the compiler tries to benefit from *concurrent* memory access wherever possible. For this, the CHC compiler needs to know the available memories and their buses. The compiler uses this information to divide data objects between these memories in such a way that it benefits the most from concurrent memory accesses. Furthermore, the compiler calculates the maximum size needed for each memory and minimizes the number of address lines needed for memory access in the final hardware assembly output.

The LSL files are generated by Altium Designer based on the schematic.

The result of the compilation phase are one or more hardware assembly files.

1.3.2. Hardware Assembly (HASM) and Assembling

Hardware Assembly, HASM in short, is a language for describing digital electronic circuits. It is the hardware equivalent of a regular assembly language.

Typically you will not read or edit HASM files as they are processed automatically in the background when you compile a project in Altium Designer. HASM files are the generated output from the hardware compiler and are further transformed by the hardware assembler. A brief description of the HASM language may however give you an understanding of the mechanisms for sharing data between code fragments that are executed by a processor core, and code fragments that are instantiated in hardware. The mechanisms to create, initialize and locate data sections are identical to the mechanisms used in traditional embedded toolsets.

HASM Language

The HASM language defines a rigid and specialized execution model that suits the needs of a compilation system that translates C into hardware description languages such as VHDL or Verilog. HASM is a higher level language than VHDL and Verilog: a shorter notation for a restricted functionality. The syntax of the language is derived from traditional assembly languages for processor cores that support instruction level parallelism. The semantics of the language enable a system to be modeled in both the functional and structural domain. The functional domain deals with basic operations such as addition and multiplication. The structural domain deals with how the system is composed of interconnected subsystems.

Traditional assembly languages model a system in the functional domain only. In HASM, the structural concepts are taken from VHDL. As a result, the translation of HASM to VHDL or Verilog is a straightforward process.

The central structural concepts in HASM are *functions* and *components*, which correspond with VHDL entities, architectures and components.

The hardware assembler **ashc** converts the HASM language into relocatable object files (ELF format). These relocatable object files are linked/located using the (multi-core) hardware linker **lkhc**. The relocatable ELF object files are passed to the hardware linker **lkhc**.

1.3.3. Linking & Locating

In a traditional software build flow the linker concatenates text, data and bss sections, and the locator places the sections at absolute addresses. When C is translated to hardware, the assemble, link and locate processes are identical to assembling, linking and locating a traditional embedded project.

The CHC compiler requires all modules to be compiled simultaneously. The compiler creates data sections in the same way a traditional embedded compiler does. The same is true for the linker **lkhc** which is basically identical to a traditional linker of an embedded toolset. You can influence the locate process using memory type specifiers and the `__at()` keyword.

See also Section 3.5, *Memory and Memory Qualifiers* .

1.3.4. HDL generation

The linked and located ELF file is transformed into a hardware design language (HDL). This is done by the HDL generator **hdlhc**. The HDL generator can generate VHDL (extension `.vhd`) and/or Verilog (extension `.v`). In Altium Designer, both formats can be generated (which might be handy when sharing files with others). The VHDL file however, is passed to Altium Designer's synthesizer and subsequently to the FPGA vendor's place-and-route tools which eventually create a bit file that can be loaded onto the FPGA.

Chapter 2. Parallelism

Understanding Parallelism

This chapter is a brief introduction on writing software for high performance processing architectures. Because a translation to hardware yields the best results if (many) instructions can be executed simultaneously, it is necessary to have a good understanding of parallelism. The issues in this chapter are independent of the execution environment: it is valid for multiple pipeline CHC but also for execution environments like DSPs or RISC processors.

The term *instruction* is associated with traditional processor cores. It defines the action that is carried out (for example: mul, div, add, …) and it's operands (for example: an immediate value, a register name, or a reference to a memory location). In this section we use the term *operation* to refer to the instructions that are executed by the electronic circuit created by the CHC compiler. An operation defines the action that is carried out (for example: mul, divide, add, …) as well as it's operands.

The CHC compiler creates small and fast electronic circuits only if the C source is *parallelizable*. So you need to understand the factors that inhibit parallelism so you can avoid them. Once operations are performed concurrently, the bandwidth of memory system that feeds data to the functional units usually becomes a bottleneck. You need to understand the issues that restrict memory bandwidth and the methods to increase memory bandwidth.

Granularity

The term *granularity* is used to indicate the size of the computations that are being performed at the same time. In the context of C-to-Hardware compilation we identify *fine grained instruction-level parallelism* (instruction level) and *coarse grained parallelism* (at thread level and process level).

Fine grained instruction level parallelism is automatically detected and exploited by the CHC compiler. Course grained parallelization is user-directed, you must explicitly specify the actions to be taken by the compiler and run-time system in order to exploit thread level and process level parallelism.

This section deals with instruction-level parallelism.

Example

Consider the following code fragment:

```
for ( int i=0; i<100; i++ )
{
    a[i] = b[i] * 2;
}
```

This loop has plenty of parallelism. If the compiler could create an electronic circuit with 100 multipliers where each multiplier accesses one member of array a and one member of b, then the loop (all 100 iterations) executes in 1 clock cycle. Whether the compiler can create such a circuit depends on the available hardware resources. Modern FPGAs provide the required number of multipliers.

However, in this example the memory system limits the amount of parallelism. If each array element was stored in a register, the compiler could construct a circuit that executes all multiplies in parallel, but normally

the array members are stored in memory. FPGAs provide multi-ported memories but the number of access ports is commonly limited to two. As a result, only one member of a and one member of b can be accessed in the same cycle. Parallelism increases if arrays a and b are stored in different dual ported memories. In that case, two members of a and two members b are accessible within the same clock cycle.

The essence of this example is that array accesses in loops impose a high load on the memory system which typically forms the bottleneck in the overall system performance.

2.1. Dependencies

In general, a C compiler translates the statements in your C program into series of low-level operations. To maintain the semantics of the program, these operations must be executed in a particular sequence. When operation A must occur before operation B we say that B depends on A, this relationship is called a *dependency*.

A compiler creates a so called *data dependency graph* that shows all dependencies for a code fragment. The data dependency graph defines the order in which operations must be executed. Dependencies inhibit parallelism. It is a task for both the software engineer and the compiler to rearrange a program to eliminate dependencies. Dependencies are commonly subdivided into *control dependencies* and *data dependencies*.

- Control dependencies arise from the control flow in the program.
- Data-dependencies arise from the flow of data between operations and occur when two operations (possibly) refer to identical storage locations (a register or a memory location).

Structural hazards (also known as *resource dependencies*) arise from the limited number of hardware resources, such as functional units and memory ports. Structural hazards inhibit parallelism but do not force a particular execution sequence.

2.1.1. Control Dependencies

A control dependency is a constraint that arises from the control flow of the application. The compiler tries to eliminate control dependencies to increase the efficiency of the generated hardware circuit.

Consider the following code fragment:

```
/*s1*/  if ( a < b ) {
/*s2*/      c = d + e;
        } else {
/*s3*/      c = d * e;
        }
```

Statement s2 and s3 have a control dependency on s1.

The electronic circuit that the compiler could create for this example, executes the compare, the addition and the multiply operations, in parallel. The output of the comparator switches a multiplexer that either assigns the result of the addition or the multiplication to variable c. This optimization is called if-conversion. The if-convertor replaces if-then-else constructs by predicated operations which can be more efficiently implemented in hardware.

Sometimes it is legal to remove predicates that are assigned as a result of an if-conversion. Consider the next code fragment:

```
/*s1*/ if ( a < b ) {
/*s2*/     c = d * e;
/*s3*/     x = c;
       }
```

Assume variable c is not used in subsequent statements. Now the compiler can remove the control dependency from s2 and schedule the multiply *before* or simultaneous with the compare. This optimization is known as *predicated operation promotion*. The predicate on s3 cannot be removed.

As the examples above show, the CHC compiler is able to translate control flow constructions quite well into efficient hardware.

2.1.2. Data Dependencies

A data dependency is a constraint that arises from the flow of data between statements/operations.

There are three varieties of data dependencies:

- *flow*, also called read-after-write (raw) dependency
- *anti*, also called write-after-read (war) dependency
- *output*, also called write-after-write (waw) dependency

The key problem with each of these dependencies is that the second statement cannot execute until the first has completed.

Flow or Read-After-Write dependency

Read-after-write dependency occurs when an operation references or reads a value assigned or written by a preceding operation:

```
/*s1*/ a = b + c;
/*s2*/ d = a + 1;
```

Anti or Write-After-Read dependency

Write-after-read dependency occurs when an operation assigns or writes a value that is used or read by a preceding operation:

```
/*s1*/ a = b + c;
/*s2*/ b = e + f;
```

Output or Write-After-Write dependency

Write-after-write dependency occurs when an operation assigns or writes a value that is also assigned by a preceding operation:

```
/*s1*/ a = b + c;
/*s2*/ a = d + e;
```

In some cases a compiler can remove anti (WAR) and output (WAW) dependencies. To do so, the compiler allocates multiple storage locations for a variable. Consider the following fragment of C code and assume that all identifiers represent scalar types with local scope.

```
/*s1*/ a = b + c;
/*s2*/ b = e + f;
/*s3*/ a = g + h;
```

When the compiler allocates different storage locations for the variable a assigned in s1 and variable a assigned in s3, the semantics of the code fragment do not change but the output (WAW) dependency between s1 and s3 is removed. When the compiler allocates different storage locations for the variable b read by s1 and assigned by s2, the anti (WAR) dependency between s1 and s2 is removed! As a result, all operations can execute concurrently.

2.1.2.1. Aliasing

Aliasing occurs when one storage location can be accessed in two or more ways. For example, in the C language the address of a variable can be assigned to a pointer and as a result, the variable's storage location is accessible via both the variable and the pointer. The variable and the pointer are aliases.

The address a pointer points to is known at run-time but can often not be computed at compile time. In such cases the compiler must assume that the pointer is an alias of all other variables. As a result all operations in which the pointer is used depend on all other operations. These extra dependencies inhibit parallelism.

Consider the following code fragments:

```
#define N 10
int a[N][N], b[N][N], d;
for ( i=0; i<N; i++ ) {
    for ( j=0; j<N; i++ ) {
        a[i][j] = a[i][j] + b[i][j] * d;
    }
}
```

This loop is parallelizable. The starting address and dimensions of the arrays are visible to the compiler, so it knows that `a[i][j]` and `b[i][j]` are not aliases.

```
#define N 10
int *a[N], *b[N], d;
for ( i=0; i<N; i++ ) {
    a[i] = (int *)malloc(N * sizeof(int));
    c[i] = (int *)malloc(N * sizeof(int));
}
for ( i=0; i<N; i++ ) {
    for ( j=0; j<N; i++ ) {
        a[i][j] = a[i][j] + b[i][j] * d;
    }
}
```

Although the loop body is identical, it is *not* parallelizable. The compiler is not allowed to make any assumptions about the pointer returned by `malloc`. Therefore the compiler cannot guarantee that `a[i][j]` and `b[i][j]` are not aliases. As a result all loop iterations execute sequentially.

```
#define N 10
void function ( int a[][N], int b[][N], int d )
{
    for ( i=0; i<N; i++ ) {
        for ( j=0; j<N; i++ ) {
            a[i][j] = a[i][j] + b[i][j] * d;
        }
    }
}
```

Whether this loop is parallelizable, depends on whether the compiler is able to deduce that a and b are not aliases. If the function is not static (thus can be called from outside the module in which it is defined), the compiler needs to analyze all modules to be able detect whether a and b alias. This global alias analysis is a time consuming process.

2.1.2.2. The `restrict` Keyword

The type qualifier `restrict` is new in ISO C99. It serves as a “no alias” hint to the compiler and can only be used to qualify pointers to objects or incomplete types. Adding the `restrict` keyword can result in great speedups at both compile-time and run-time.

By definition, a `restrict` qualified pointer points to a storage location that can only be accessed via this pointer; no other pointers or variables can refer to the same storage location.

The ISO C99 standard provides a precise mathematical definition of `restrict`, but here are some common situations:

- A `restrict` pointer which is a function parameter, is assumed to be the only possible way to access its object during the function’s execution. By changing the function prototype in the previous example from:

  ```
  function ( int a[][N], int b[][N], int d )
  ```

  to:

  ```
  function ( int a[restrict][N], int b[restrict][N], int d )
  ```

  Alternative syntax:

  ```
  function ( int **restrict a, int **restrict b, int d )
  ```

  The loop body becomes parallelizable, without relying on the results of global alias analysis.

- A file-scope pointer declared using `restrict` is assumed to be the only possible way to access the object to which it refers. This may be an appropriate way to declare a pointer initialized by `malloc` at run time.

```
extern int * restrict ptr_i;

void init ( void )
{
 ptr_i = malloc(20 * sizeof(int));
}
```

2.1.2.3. Loop Carried Dependencies

The notion of data dependency is particularly important for loops where a single misplaced dependency can force the loop to be run sequentially.

To execute a loop as fast as possible, the operations within the loop body as well as multiple iterations of the loop should execute in parallel. Multiple iterations of the loop body can execute in parallel if there are no data dependencies between the iterations.

A dependency may be loop-independent (i.e. independent of the loop(s) surrounding it), or loop-carried. Loop-carried dependencies result from dependencies among statements that involve subscripted variables which nest inside loops. The dependency relationships between subscripted variables become much more complicated than for scalar variables, and are functions of the index variables, as well as of the statements.

In the code fragment below the anti dependency caused by writing `b[i][j]` in s4 and reading `b[i][j]` in s3 is loop-independent. The flow dependency of s4 on s3, arising from setting an element of `a[]` in s3 and s4's use of it one iteration of the inner `j` loop later, is loop-carried, in particular carried by the inner loop.

```
/*s1*/ for (int i=0; i<3; i++) {
/*s2*/     for (int j=0; j<4; j++) {
/*s3*/         a[i][j] = b[i][j] + c[i][j];
/*s4*/         b[i][j] = a[i][j-1] * d[i+1][j] + t;
           }
       }
```

Try to avoid loop-carried dependencies by restructuring your source code.

2.2. The Memory System

If many operations execute in parallel, the memory system should provide the necessary bandwidth to feed all operations with data. Typically the performance of the memory system restricts overall system performance. Especially if a processor core shares data with hardware functions, that shared memory will likely be the system's bottleneck.

Registers

Programmable hardware (FPGA's) offers a virtually unlimited amount of registers to store variables and temporary results. Registers are the fastest accessible storage locations. All registers can be accessed in parallel.

FPGA memory

Modern FPGAs supply large quantities of configurable on-chip block-RAM and distributed RAM. The characteristics of the on-chip RAM such as the number of access ports, the bit width and size of the RAMs are configured when the bit-file is loaded into the FPGA. Variables stored in different or in multi-ported RAMs can be accessed in parallel.

Memory latency

Memory latency is defined as the time it takes to retrieve data from memory after the data request. Typically on-chip memory has a latency of one, which means that the data arrives in the clock cycle following the request. The memory latency can also be variable; in that case a handshake signal is asserted once the data becomes available.

If multiple components (processor core, hardware function or peripheral) share memory via a common bus interface the memory latency will be variable. Variable latencies have a negative effect on system performance, program execution halts until the handshake signal is asserted.

How the compiler uses registers and memory

The compiler tries to construct a high-performance memory system based on the characteristics of both the source code and the targeted FPGA device. The FPGA device characteristics are defined in a *resource definition file* and in an *linker description language file* (lsl file).

The compiler allocates variables whose address is not taken in registers.

Large data structures, arrays, and variables whose address is taken, are allocated in memory.

The compiler analyzes variable access and pointer dereference patterns. Based on this analysis the variables and pointers are grouped in parallel accessible clusters. Data objects allocated in different clusters can be accessed concurrently. Clusters that contain data that cannot be accessed from outside the hardware block are mapped to either distributed RAM or to block-RAM. The compiler instantiates this memory. If sufficient memory resources are available, each memory contains only one cluster, and the most frequently accessed clusters are located in multi-ported memories. The remaining clusters are located into a memory that is shared with the processor core's address space.

How to improve the compiler's default allocation

Given the characteristics of the source code the compiler may not be able to construct an efficient memory system. Aliasing may inhibit the instantiation of multiple parallel accessible memories.

The ISO standardization committee has introduced C language extensions to support multiple address spaces. Named address spaces are implemented by memory type qualifiers in C declarations. These qualifiers associate a variable with a specific address space. Named address space support can be provided within the current C standards by the single addition of a memory type qualifier in variable declarations. SeeSection 3.5, *Memory and Memory Qualifiers* for more information about memory type qualifiers.

Scheduling and operation chaining

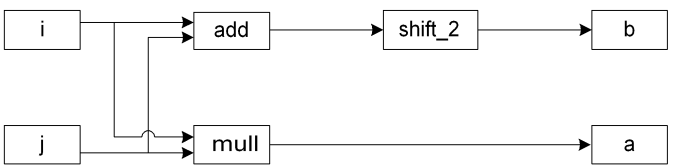
The main inputs for the compiler's scheduler are the data dependency graph and the resource definition file. The data dependency graph defines the order in which operations are allowed to be executed. The resource definition file describes the available number of functional units and their latencies. Based upon the latencies the compiler decides whether it can chain operations. If two operations are chained they execute within the same cycle and the output of the first operation is the input for the second.

Consider the following code fragment:

```
/*s1*/ a = i * j;
/*s2*/ b = i + j;
/*s3*/ b = b << 2;
```

There is one dependency, statement s3 must execute after s2, the other statements may execute concurrently. The latencies of the operations are commonly quite different. Assume the following latencies are defined in the resource definition file: multiply 50, addition 30, and shift 5 time units. Notice that a shift operation with a constant shift value consumes virtually zero time since there is no logic involved. Given these latencies the shift operation can be chained with the addition and the chained operation executes concurrently with the multiply.

Figure 2.1. Scheduling



Chapter 3. C Language Implementation

This chapter describes the CHC specific features of the C language, including language extensions that are not defined in ISO-C. The ISO-C standard defines the C language and the C libraries. The CHC compiler supports the ISO-C language and provides additional features to program the special functions of the target. The C library implementation is comparable to the C library of compilers for embedded systems.

This chapter sometimes refers to CHC compiler options. Although it is rarely necessary to set CHC compiler options by hand, in Section 4.1.1, *CHC Compiler Options* * it is explained how you can access and set CHC compiler options in Altium Designer.

The following C language features implemented in the compiler deviate from the ISO-C standard:

- Function prototypes: function prototypes are mandatory instead of optional
- Hardware functions are not reentrant:
  - Recursion is not possible: cycles in the call graph are not allowed
  - Hardware functions cannot be called concurrently from multiple processes (for example by both the main program and an interrupt handler)
- Data types: `double` precision floating point data can be treated as single precision float with the option **--no-double** (**-F**)
- Type specifiers: `_Complex` and `_Imaginary` are not implemented

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

- Operating modes for ISO-C99, ISO-C90, and GNU gcc compatibility
- Keywords to specify which functions should be compiled to hardware
- Keywords to specify the calling convention and bus interface of the hardware functions
- Keywords to specify memory types for data
- Keywords to optimize the hardware output
- An attribute to locate data at absolute addresses
- Pragmas to control the compiler from within the C source
- Predefined macros

All non-standard keywords have two leading underscores (__).

3.1. Data Types

The C compiler supports the ISO C99 defined data types. The compiler can operate in either 32-bit mode, which is the default mode, or in 16-bit mode, depending on the embedded compiler it is used with. The sizes of all data types are shown in the following tables.

- Size (mem) lists the size of the object when loaded in memory.
- Size (reg) lists the size of the object when loaded in a register.

The CHC compiler tries to minimize the size of a data object based on the use of the variable in the source code, thus saving the number of flip-flops and number of wires that need to be instantiated in the hardware.

Data types in 32–bit mode

C Type	Size (mem)	Size (reg)	Align	Limits
_Bool	8	1	8	0 or 1
signed char	8	≤ 8	8	[-0x80, +0x7F]
unsigned char	8	≤ 8	8	[0, 0xFF]
short	16	≤ 16	16	[-0x8000, +0x7FFF]
unsigned short wchar_t	16	≤ 16	16	[0, 0xFFFF]
int	32	≤ 32	32	[-0x80000000, +0x7FFFFFFF]
unsigned int	32	≤ 32	32	[0,0xFFFFFFFF]
enum *)	8 8 32	1 ≤ 8 ≤ 32	8 8 32	0 or 1 [-0x80, +0x7F] [-0x80000000, +0x7FFFFFFF]
long ptrdiff_t	32	≤ 32	32	[-0x80000000, +0x7FFFFFFF]
unsigned long size_t	32	≤ 32	32	[0,0xFFFFFFFF]
long long	64	≤ 64	32	[-0x8000000000000000, +0x7FFFFFFFFFFFFFFF]
unsigned long long	64	≤ 64	32	[0, 0xFFFFFFFFFFFFFFFF]
float (23-bit mantissa)	32	32	32	[-3.402E+38, -1.175E-38] [+1.175E-38, +3.402E+38]
double long double (52-bit mantissa)	64	64	32	[-1.798E+308, -2.225E-308] [+2.225E-308, +1.798E+308]
pointer	32	≤ 32	32	[0,0xFFFFFFFF]

Data types in 16–bit mode (with compiler option --integer-16bit)

C Type	Size (mem)	Size (reg)	Align	Limits
_Bool	8	1	8	0 or 1
signed char	8	≤ 8	8	[-0x80, +0x7F]
unsigned char	8	≤ 8	8	[0, 0xFF]
short	16	≤ 16	16	[-0x8000, +0x7FFF]
unsigned short __wchar_t	16	≤ 16	16	[0, 0xFFFF]
int	16	≤ 16	16	[-0x8000, +0x7FFF]
unsigned int	16	≤ 16	16	[0,0xFFFF]
enum *)	8 8 32	1 ≤ 8 ≤ 32	8 8 32	0 or 1 [-0x80, +0x7F] [-0x80000000, +0x7FFFFFFF]
long __ptrdiff_t	32	≤ 32	32	[-0x80000000, +0x7FFFFFFF]
unsigned long __size_t	32	≤ 32	32	[0,0xFFFFFFFF]
long long	64	≤ 64	32	[-0x8000000000000000, +0x7FFFFFFFFFFFFFFF]
unsigned long long	64	≤ 64	32	[0, 0xFFFFFFFFFFFFFFFF]
float (23-bit mantissa)	32	32	32	[-3.402E+38, -1.175E-38] [+1.175E-38, +3.402E+38]
double long double (52-bit mantissa)	64	64	32	[-1.798E+308, -2.225E-308] [+2.225E-308, +1.798E+308]
pointer	32	≤ 32	32	[0,0xFFFFFFFF]

*) When you use the `enum` type, the compiler uses the smallest possible type (`char`, `unsigned char` or `int`) to represent the value.

3.2. Predefined Preprocessor Macros

The CHC compiler supports the predefined macros as defined in the table below. The macros are useful to create conditional C code.

Macro	Description
__BIG_ENDIAN__	Expands to 1 if the compiler operates in big endian mode. Otherwise not recognized as a macro.
__LITTLE_ENDIAN__	Expands to 1 if the compiler operates in little endian mode. Otherwise not recognized as a macro.

Macro	Description
__BUILD__	Identifies the build number of the compiler, composed of decimal digits for the build number with leading zeros removed, three digits for the major branch number and three digits for the minor branch number. For example, if you use build 1.22.1 of the compiler, __BUILD__ expands to 1022001. If there is no branch number, the branch digits expand to zero. For example, build 127 results in 127000000.
__CHC__	Identifies the compiler. You can use this symbol to flag parts of the source which must be recognized by the **chc** compiler only. It expands to 1.
__DATE__	Expands to the compilation date: “mmm dd yyyy”.
__DOUBLE_FP__	Expands to 1. Reserved macro.
__DSPC__	Indicates conformation to the DSP-C standard. It expands to 0.
__FILE__	Expands to the current source file name.
__LINE__	Expands to the line number of the line where this macro is called.
__INTEGER_16BIT__	Expands to 1, if the compiler operates in 16-bit mode.
__REVISION__	Expands to the revision number of the compiler. Digits are represented as they are; characters (for prototypes, alphas, betas) are represented by -1. Examples: v1.0r1 -> 1, v1.0rb -> -1
__SINGLE_FP__	Unrecognized as macro.
__STDC__	Identifies the level of ANSI standard. Expands to 0.
__STDC_HOSTED__	Always expands to 0, indicating the implementation is not a hosted implementation.
__STDC_VERSION__	Identifies the ISO-C version number. Expands to 199901L for ISO C99 or 199409L for ISO C90.
__TASKING__	Identifies the compiler as a TASKING compiler. Expands to 1.
__TIME__	Expands to the compilation time: “hh:mm:ss”
__VERSION__	Identifies the version number of the compiler. For example, if you use version 2.1r1 of the compiler, __VERSION__ expands to 2001 (dot and revision number are omitted, minor version number in 3 digits).
__TSK3000__ __C3000__	Expand to 1 if the CHC is compatible with the TSK3000 processor and compiler. These macros are automatically set by Altium Designer.
__PPC__ __CPPC__	Expand to 1 if the CHC is compatible with the Power PC processor and compiler. These macros are automatically set by Altium Designer.
__ARM__ __CARM__	Expand to 1 if the CHC is compatible with the ARM processor and compiler. These macros are automatically set by Altium Designer.
__CMB__	Expands to 1 if the CHC is compatible with the MicroBlaze processor and compiler. This macro is automatically set by Altium Designer.

Macro	Description
__NIOS2__ __CNIOS__	Expand to 1 if the CHC is compatible with the NIOS processor and compiler. These macros are automatically set by Altium Designer.

Example

```
#if __CHC__
/* this part is only for the C-to-Hardware compiler */
...
#endif
```

3.3. Pragmas to Control the Compiler

Pragmas are keywords in the C source that control the behavior of the compiler. Pragmas overrule compiler options.

When the C-to-Hardware compiler is used in combination with an embedded compiler, the C source code is both processed by the embedded compiler and the CHC compiler. It depends on the available options of the embedded compiler whether certain pragmas are recognized or have any relevance. For example, **#pragma unroll_factor** is only recognized by the embedded compiler if the embedded compiler supports the optimization *loop unrolling*. If not, the embedded compiler does not recognize the pragma and will ignore it. The pragmas described below are the CHC compiler pragmas. Refer to the Users Guide of the embedded compiler to see which options and pragmas it supports.

The general syntax for pragmas is:

`#pragma` *`pragma-spec pragma-arguments`* `[ON | OFF | DEFAULT | RESTORE]`

or:

`_Pragma( "`*`pragma-spec pragma-arguments`* `[ON | OFF | DEFAULT | RESTORE]" )`

Pragmas marked with (*) accept the following special arguments:

`default`	set the pragma to the initial value
`restore`	restore the previous value of the pragma

Pragmas marked with (+) are boolean flags, and accept the following arguments:

`on`	switch the flag on (same as without argument)
`off`	switch the flag off

The compiler recognizes the following pragmas, other pragmas are ignored.

#pragma alias *symbol=defined_symbol*

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

#pragma extern *symbol*

Force an external reference (**.extern** assembler directive), even when the *symbol* is not used in the module.

#pragma inline
#pragma noinline
#pragma smartinline

Instead of the qualifier `inline`, you can also use pragma `inline` and pragma `noinline` to inline a function body:

```
int  w,x,y,z;

#pragma inline
int add( int a, int b )
{
     int i=4;
     return( a + b );
}
#pragma noinline

void main( void )
{
     w = add( 1, 2 );
     z = add( x, y );
}
```

If a function has an `inline` or `__noinline` function qualifier, then this qualifier will overrule the current pragma setting.

By default, small functions which are not called from many different locations, are inlined. This reduces execution speed at the cost of area. With the pragma `noinline` / pragma `smartinline` you can temporarily disable this optimization.

#pragma macro
#pragma nomacro (*) (+)

Enable or disable macro expansion.

#pragma message "*message*" ...

Print the message string(s) on standard output. Arguments are first macro expanded.

#pragma optimize *flags* (*)
#pragma endoptimize

You can overrule the default compiler optimization for the code between the pragmas `optimize` and `endoptimize`.

The pragma uses the following flags that are similar to compiler options for **Optimization** in embedded toolsets:

+/-coalesce	**a/A**	Coalescer: remove unnecessary moves
+/-outparams	**b/B**	Consider pointer access to function parameters as return value(s) and pass them via registers if possible. This optimization is *always* performed for C code symbols, even when the **#pragma optimize 0** or compiler option **--optimize=0** is set. (See also symbol qualifier Section 3.4.1, “__out”)
+/-cse	**c/C**	Common subexpression elimination
+/-predicate	**d/D**	Predicate optimizations
+/-expression	**e/E**	Expression simplification
+/-flow	**f/F**	Control flow simplification
+/-glo	**g/G**	Generic assembly code optimizations
+/-inline	**i/I**	Automatic function inlining
+/-schedule	**k/K**	Instruction scheduler
+/-loop	**l/L**	Loop transformations
+/-simd	**m/M**	Perform simd optimization
+/-forward	**o/O**	Forward store
+/-propagate	**p/P**	Constant propagation
+/-speculate	**q/Q**	Speculate
+/-subscript	**s/S**	Subscript strength reduction
+/-tree	**t/T**	Expression tree reordering
+/-unroll	**u/U**	Unroll small loops
+/-ifconvert	**v/V**	Convert IF statements using predicates
+/-pipeline	**w/W**	Software pipelining
+/-peephole	**y/Y**	Peephole optimizations

Use the following options for predefined sets of flags:

0	No optimization Alias for **ABCDEFGIKLMOPQSTUVWY**

No optimizations are performed. The compiler tries to achieve an optimal resemblance between source code and produced code. Expressions are evaluated in the same order as written in the source code, associative and commutative properties are not used.

1	Few optimiziations Alias for **aBcDefgIKLMOPqSTUVWY**

The generated circuit is still comprehensible and could be manually debugged.

2	Release purpose optimizations Alias for **abcdefglklmopqstUvwy**

Enables more optimizations to reduce area and/or execution time. The relation between source code and generated circuit may be hard to understand. This is the default optimization level.

3	Aggressive (all) optimizations Alias for **abcdefgiklmopqstuvwy**

Enables aggressive global optimization techniques. The relation between source code and generated instructions is complex and hard to understand. Inlining (i) and loop unrolling (u) are enabled. These optimizations enhance execution time at the cost of extra generated hardware.

#pragma source (*)(+)
#pragma nosource

With these pragmas you can choose which C source lines must be listed as comments in assembly output.

#pragma stdinc (*)(+)

This pragma changes the behavior of the #include directive. When set, the options **-I** and **-no-stdinc** of the embedded compiler are ignored.

#pragma tradeoff *level* (*)

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

#pragma unroll_factor *number* (*)

With this pragma you can specify a unroll factor if you have set **#pragma optimization +unroll**. The unroll factor determines to which extent loops should be unrolled. Consider the following loop:

```
for ( i = 1; i < 10; i++ )
{
    x++;
}
```

With an unroll-factor of 2, the loop will be unrolled as follows:

```
for (i = 1; i < 5; i++ )
{
    x++;
    x++;
}
```

If you enable the unroll optimization, but do not specify an unroll factor, the compiler determines an unroll factor by itself.

#pragma warning [*number*,...] (*)

With this pragma you can disable warning messages. If you do not specify a warning number, all warnings will be suppressed. This pragma works the same as the **--no-warning** option of an embedded compiler.

#pragma weak *symbol*

Mark a symbol as "weak". The symbol must have external linkage, which means a global or external object or function. A static symbol cannot be declared weak.

A weak external reference is resolved when a global (or weak) definition is found in one of the source files. However, a weak reference will not cause the extraction of a module from a library to resolve the reference. When a weak external reference cannot be resolved, the null pointer is substituted.

A weak definition can be overruled by a normal global definition. The linker will not complain about the duplicate definition, and ignore the weak definition.

3.4. Function and Symbol Qualifiers

3.4.1. Compiling to Hardware

To compile C source code to hardware in Altium Designer, the schematic must contain either an Application Specific Processor (WB_ASP) component or a C code symbol.

The CHC compiler supports a number of function qualifiers and memory qualifiers that specify whether and how C source is compiled to hardware. You can manually add these function qualifiers and/or symbol qualifiers in your C source, but Altium Designer adds them automatically when you are filling out the dialogs for the Application Specific Processor or C code symbol.

The following function qualifiers are implemented:

__rtl

With the `__rtl` function qualifier you tell the compiler to compile this function to hardware.

An `__rtl` qualified function can be called by other `__rtl` qualified functions but cannot be called by non `__rtl` qualified functions. To make an `__rtl` qualified function callable by a non `__rtl` qualified function, you must also use the function qualifier `__export`.

__export

An `__export` qualified function is callable from the processor core, that is, it can be called from non `__rtl` qualified functions. The `__export` qualifier must be used in combination with the `__rtl` and `__CC()` qualifiers. The `__export` qualifier causes the function's interface signals to be included in the top-level entity definition of the generated VHDL or Verilog code.

Application Specific Processor

In the configurations dialog of the WB_ASP component you can select which functions should be compiled to hardware:

1. Right-click on the ASP component and select **Configure ... (WB_ASP)...**

   The Configure (WB_ASP Properties) dialog appears.

2. On the right side of the dialog you'll find the section **Symbols In Hardware**.

   - In the lower part, in the column **Implement in Hardware**, select the functions that you want to be compiled to hardware. (This is the equivalent of `__rtl` qualifier.)

   - If the hardware function should be callable from a software function, mark it as exported as well in the column **Export to Software**. (This is the equivalent of the `__export` qualifier.)

C code symbol

The C code symbol is associated with a C source document that contains the function to be compiled. This 'main' function may call other functions. For the 'main' function the `__export` qualifier is automatically added. Because all functions in the C source belonging to the C code symbol are compiled to hardware by definition, an `__rtl` qualifier is not needed in this context.

`__`import

An `__import` qualified function is a function, defined in a different project, which is callable from an `__rtl` qualified function in the current project. To interface with an `__import` qualified function, the compiler adds the required signals to the top level entity. The `__import` qualifier can only be used in combination with the `__rtl` qualifier. The imported function must have been defined elsewhere.

`__`CC (*bus*, [*id*], [nowait], [combinatorial])

For each function that should be compiled to hardware, Altium Designer automatically adds the `__CC` (calling convention) qualifier with fixed attributes. It not necessary, but possible to add this qualifier to your C source by hand.

`__CC` is the calling convention qualifier. The `__CC` qualifier must be used in combination with the `__rtl` and `__export` qualifiers and has the following syntax:

`__CC (`*`bus`*`,[`*`id`*`],[nowait],[combinatorial])`

bus	Specifies the interconnect mechanism between the processor core and the hardware function. Possible values are: `wishbone` : interface via Wishbone bus `nios_ci` : interface via NIOS II custom instruction interface `parallel` : default, no bus specific wrapper is created
id	Defines how the hardware function is connected to/addressed by the processor core. The *id* must be a unique number within the application scope, and should be in the range 0..31.

`nowait`	The optional `nowait` parameter indicates that the processor core will not wait until the hardware function returns, but causes the processor core to proceed immediately with execution of the instruction(s) following the call to the hardware function. As a result the processor core and the hardware functions run in parallel. This parameter is only possible in combination with a `wishbone` bus.
`combinatorial`	With this parameter you ask the CHC compiler to compile the C source to a combinatorial circuit (or: *zero-cycle* circuit as opposed to a *multi-cycle* circuit which needs a clock, reset, start and done signal on the schematic). If the compiler is not able to generate a zero-cycle schedule, it issues an error. This parameter is only possible in combination with a `parallel` bus.

Application Specific Processor example:

```
__rtl __export __CC(wishbone, 1) void my_hw_func ( void );
```

Because of the `__rtl` qualifier, the code for the C function `my_hw_func` is generated by the CHC compiler.

The `__export` qualifier enables the function `my_hw_func` to be called by code that executes on the processor core (by non `__rtl` qualified functions).

Because of the `__CC` qualifier, the CHC compiler produces a Wishbone bus interface which connects an ASP to a soft-core processor. Because the `nowait` parameter is not specified, the processor core will wait (by entering a polling loop) until `my_hw_func` returns before it executes subsequent instructions.

Altium Designer generates this function qualifier automatically with the `wishbone` attribute, a unique `id` and without the `nowait` attribute:

```
__CC(wishbone, 1)
```

You can verify this by opening the qualifier file *`projectname`*`.qua` which is located in the output directory of your embedded project.

C code symbol example:

```
__export __CC(parallel, combinatorial)
void add(int8_t a, uint8_t b, __out int16_t* sum )
```

The `__rtl` qualifier is not needed because all C source for C code symbols is compiled to hardware.

The `__export` qualifier is added to the 'main' function and enables access to the (hardware compiled) function (the input and output ports on the schematic).

Because of the `__CC` qualifier, the CHC compiler produces a parallel bus interface through which you can connect the the input and output ports of the C code symbol to other components.

`__`width()

When programming in C, input and output parameters of functions are defined by their types (char, int, ...). Each data type has a specified number of bits and this determines the number of lines the CHC compiler will generate when it compiles the function to hardware. Especially for C code symbols, you may

want to be more specific in cases where you know the maximum number of bits needed for an input or output parameter.

With the symbol qualifier `__width(x)` you can specify the number of bits for a symbol.

Example

Suppose you have a C code symbol that is programmed with the following function:

```
void add(int8_t a, int8_t b, int16_t* result)
{
 result* = a + b;
}
```

The function has two input parameters with a width of 8 bits and one output parameter with a width of 16 bits. You can restrict the number of bits for the input and/or output parameters as follows:

```
void add(int8_t __width(6) a, int8_t __width(6) b, int16_t __width(7)* result)
```

Be aware of the following restrictions:

- You can use the `__width` qualifier only for integral types
- The specified width should be in the range [1 .. 64].
- The specified width cannot exceed the width of the type: `__width(9) uint8_t obj` causes an error.

In Altium Designer you can specify the bit-width from the C code symbol on your schematic:

1. Double-click on a port on the C code symbol

   The C code Entry (Parameter) dialog appears.

2. Enter the number of bits in the **Integer Width** field.

__out

With the symbol qualifier `__out` you force the CHC compiler to treat a function parameter with pointer type as an extra return value. In this case, the compiler tries to pass the variable via registers. If this is not possible, the compiler issues an error.

This type of optimization is always done for C code symbols (the CHC compiler implicitly places the symbol qualifier `__out` before every parameter with pointer type).

The optimization will fail when you try to write to the pointer offset or when you also read the indirected pointer:

```
void add(int8_t a, uint8_t b, __out int16_t* sum )
{
    *(sum+1) = 3;   // optimization fails: writing to pointer offset
    *sum += a + b;  // optimization fails: reading from indirected pointer
}
```

Restrictions on Function Calls

- The C-to-Hardware compiler does not support function reentrancy, so recursion (cycles in the call graph) and concurrent access by multiple processes are not allowed. In case you call a hardware function from an interrupt handler, make sure all concurrency conflicts are solved.
- For each hardware function a function prototype that declares the return type of the function and also declares the number and type of the function's parameters is required. Functions with a variable number of arguments are supported.
- The following rules apply to the interaction between software and hardware:
  - Recursion and concurrent access by multiple processes are only allowed with non `__rtl` qualified functions in the call graph.
  - An `__export __rtl` qualified function can be called from non `__rtl` qualified functions, thus can be called from functions that are executed by a processor core.
  - An `__export __rtl` qualified function can be called from `__rtl` qualified and `__rtl __export` qualified functions.
  - An `__rtl` qualified function can be called from `__rtl` qualified functions.
  - An `__rtl` qualified function cannot call a non `__rtl` qualified function, so a function that is converted into an electronic circuit cannot call a function that is executed by a processor core.
  - Non `__rtl` qualified functions can be inlined into `__rtl` qualified functions.
  - An `__export` qualified function cannot be inlined.
  - Function pointers are not allowed as argument or return value in `__export` qualified functions.
  - An `__export` qualified function cannot have a variable number of arguments.

3.4.2. Inlining Functions: inline / __noinline

During compilation, the C compiler automatically inlines small functions in order to reduce interconnect overhead (smart inlining). The compiler inserts the function body at the place the function is called. If the function is not called at all, the compiler does not generate code for it. The C compiler decides which functions will be inlined. You can overrule this behavior with the two keywords `inline` (or `__inline` in C90 mode) and `__noinline`.
You can also use this qualifier in combination with the `__rtl` qualifier (assigned within Altium Designer, as described previously). The inlined function then is integrated in the same hardware component as its caller.

With the `inline` keyword you force the compiler to inline the specified function, regardless of the optimization strategy of the compiler itself:

```
inline unsigned int abs(int val)
{
    unsigned int abs_val = val;
    if (val < 0) abs_val = -val;
```

```
    return abs_val;
}
```

You must define inline functions in the same source module in which you call the function, because the compiler only inlines a function in the module that contains the function definition. When you need to call the inline function from several source modules, you must include the definition of the inline function in each module (for example using a header file).

With the `__noinline` keyword, you prevent a function from being inlined:

```
__noinline unsigned int abs(int val)
{
    unsigned int abs_val = val;
    if (val < 0) abs_val = -val;
    return abs_val;
}
```

Using pragmas: inline, noinline, smartinline

Instead of the `inline` qualifier, you can also use `#pragma inline` and `#pragma noinline` to inline a function body:

```
#pragma inline
unsigned int abs(int val)
{
    unsigned int abs_val = val;
    if (val < 0) abs_val = -val;
    return abs_val;
}
#pragma noinline
void main( void )
{
    int i;
    i = abs(-1);
}
```

If a function has an `inline`/`__noinline` function qualifier, then this qualifier will overrule the current pragma setting.

With the `#pragma noinline` / `#pragma smartinline` you can temporarily disable the default situation that the C compiler automatically inlines small functions.

3.5. Memory and Memory Qualifiers

3.5.1. Introduction

Traditional processor cores provide a fixed set of hardware resources which are described in the processor's data books. High performance processor cores –including most signal processors– implement multiple memory spaces to enable the processor core to access multiple data objects within a single clock cycle. Parallel access to data is required to keep the processor core's functional units busy.

The ISO-C language abstracts a processor core's memory system as one large array of memory, so language extensions had to be introduced to provide better programming support for these high-performance processor cores. So-called memory type qualifiers allow you to explicitly declare the memory space in which a data object is allocated. If a pointer is dereferenced, the offset and the memory space the pointer points to, must be known. Therefore the memory space to which the pointer points, is also part of the pointer's type specifier.

Language extension for memory space qualifiers have been standardized see:

- DSP-C an Extension to ISO/IEC 9899:1999(E) PROGRAMMING LANGUAGES C
- ISO TR18037 Technical Report on Extensions for the Programming Language C to support embedded processors

The Altium Viper compilers comply to the DSP-C specification which is a predecessor of the ISO technical report.

Resource File and LSL file

A hardware compiler creates an execution environment (in contrast to an embedded compiler which generates an instruction sequence that is executed by a processor core). The hardware resources available to the CHC compiler are described in the *resource definition file* while the *linker script language file* (LSL file) describes the memories available to the CHC compiler.

The resource definition file describes the available number of functional units and their characteristics that the compiler can instantiate. The LSL file describes the available number of memories and their characteristics that the compiler can use and/or instantiate. Common FPGA devices contain a lot of local on-chip RAM, known as *block-RAM* or *distributed RAM*. Programmable hardware offers an abundance of functional units and as a result, the memory system is often the performance bottleneck. To solve this problem, the compiler creates a memory system that supports concurrent memory accesses. For this purpose the CHC compiler supports up to 10 different memory spaces.

The compiler automatically distributes data objects over multiple memory spaces. Only if the compiler is not able to create a memory partitioning that satisfies your performance requirement, it is necessary to use memory type qualifiers. It may also be useful to use memory space qualifiers to explicitly qualify variables and/or pointers that are located in / point-to the memory that is shared with the processor core.

Initialized Variables with Static Storage

The memory on the FPGA system is initialized at system configuration time, when the bit-file that configures the FPGA is loaded into the FPGA. Variables located in memory that is instantiated by the CHC compiler (these memories are components under the top-level VHDL entity), are initialized.

3.5.2. Storage Class Specifier: __rtl_alloc

It is possible to indicate that a static data object (variable) should be allocated by the CHC compiler instead of the embedded compiler. The CHC compiler allocates data objects in one of the block RAMs on the ASP where it can be accessed much faster than when allocated by the compiler, outside the ASP. `__rtl_alloc` qualified variables can only be accessed by `__rtl` qualified functions. If you try to access an `__rtl_alloc` qualified variable from a non `__rtl` qualified function, the compiler issues a error.

You can assign the storage class specifier `__rtl_alloc` to data objects with static storage and a scope other then block scope. `__rtl_alloc` qualified data objects should not be larger than 32 kB while by default, a total of 96 kB of ASP block RAM is available for `__rtl_alloc` qualified data objects.

Example

```
int sw_var;  int __rtl_alloc hw_var;
```

The storage for `sw_var` is allocated by the embedded toolset; the storage for `hw_var` is allocated by the CHC compiler.

Like the `__rtl` qualifier, you can use Altium Designer to mark these variables from the ASP configuration dialog instead of using the `__rtl_alloc` qualifier:

1. Right-click on the ASP component and select **Configure ... (WB_ASP)...**

   The Configure (WB_ASP Properties) dialog appears.

2. On the right side of the dialog you'll find the section **Symbols in Hardware**.

   - In the upper part, in the column **Allocate in Hardware**, select the variables that you want to be located by the hardware compiler. (This is the equivalent of `__rtl_alloc` qualifier.)

3.5.3. Memory Qualifier: __mem0 .. __mem9

Memory type qualifiers are used to specify the memory space in which a data object is allocated or to specify the memory space to which a pointer points.

The CHC compiler treats every physical memory defined in the LSL file as a memory space. Each memory in the LSL file is associated with one of the memory type qualifiers `__mem0` .. `__mem9`, depending on the order in which they appear in the LSL file. For performance reasons, the LSL file specifies only `__mem0`, `__mem1__mem2`, and `__mem3`. It is possible to manually edit the LSL file `memory_asp.lsl` which is located in the output directory of your project, and specify more memories up to the supported maximum of 10.

Syntax

`__mem`*X*

where *X* is a digit in the range [0..9].

If you do not specify a memory qualifier, the default `__mem0` qualifier is assumed. This qualifier is associated with shared memory, whereas the other qualifiers `__mem1..__mem9` are associated with block RAMs on the ASP.

Limiting conditions

- A list of declaration type specifiers cannot contain different memory type qualifiers.
- Structure or union members cannot have memory qualifiers.

Semantics

If the same memory qualifier appears more than once in the same specifier-qualifier-list (either directly or via one or more typedefs), the behavior is the same as if it appeared only once. The memory space qualified with `__mem0`, is the shared memory space.

The other memory spaces `__mem1`, `__mem2`, ... are typically mapped to the available data memory in descending order of parallel accessibility. This means that parallel use of `__mem1` and `__mem2` is equally or better possible than parallel use of `__mem2` and `__mem3`, and so on. A `__mem`*X* qualified pointer cannot be converted to a pointer without a type qualifier, or with a different type qualifier, or vice versa.

A conforming implementation may map memory qualified objects with automatic storage duration to default memory space.

No assumptions can be made when casting a pointer to a different memory space. Such casts are therefore not allowed.

Additional constraints to the relational operators concerning memory qualified operands:

- Both operands are pointers with equal memory qualifiers.

Additional constraints to the equality operators concerning memory qualified operands:

- Both operands are pointers with equal memory qualifiers.

Each memory space in the LSL file has a user defined name; you can use this name as an alias for the `__mem`*X* qualifier. By default, the user names correspond to the qualifier name. You can map the predefined memory type qualifier `__mem`*X* to an arbitrary string by editing the LSL file. For example, if you share memory between a processor core and the hardware functions, you may prefer to use qualifier `__shared_mem` instead of `__mem`*X* to increase the readability of the source code. These LSL names should not conflict with C keywords and symbols used in your C source.

Example

```
int __mem1 gi;

void func ( void )
{
 static int __mem1 * __mem0 pi = &gi;
}
```

Variable `gi` is located in `__mem1`. Pointer `pi` is located in `__mem0` and points to a location in `__mem1`. Now it is legal to assign the address of `gi` to `pi`. The pointer is located in shared memory and is accessible both from `__rtl` qualified functions and non `__rtl` qualified functions.

See also Section 3.5.5, *Shared Memory*.

3.5.4. Placing a Data Object at an Absolute Address: __at()

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

With the attribute `__at()` you can specify an absolute address. The address is a 32-bit address.

Examples

```
unsigned char Display[80*24] __at( 0x2000 );
```

The array `Display` is placed at address 0x2000.

```
int i __at(0x1000) = 1;
```

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

Limiting conditions

Take note of the following limiting conditions if you place a variable at an absolute address:

- The argument of the `__at()` attribute must be a constant address expression.
- You can place only variables with static storage at absolute addresses. Parameters of functions, or automatic variables within functions cannot be placed at absolute addresses.
- You cannot place structure members (in contrast to the *whole* structure) at an absolute address.
- Absolute variables cannot overlap each other.
- When you declare the same absolute variable within two modules, this produces conflicts (except when one of the modules declares the variable 'extern').
- If you use 0 as an address, the value is ignored. A zero value indicates a relocatable section.

3.5.5. Shared Memory

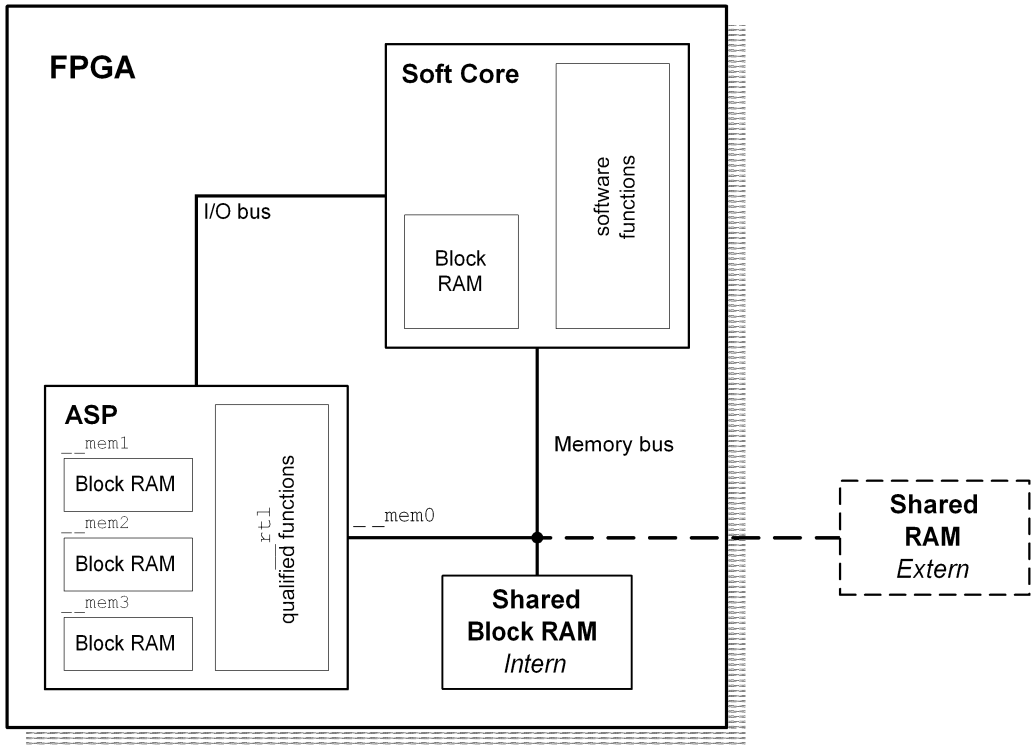
An FPGA design may contain both software functions (programmed soft processor core) and hardware functions (an ASP). This inevitably creates the need for sharing data between hardware and software. The figure below shows a schematized design, containing a soft-core, an application specific processor (ASP) and several memory locations. See also the FPGA design created in the tutorial *TU0130 Getting Started with the C-to-Hardware Compiler*.

Memory on an FPGA, called block RAM, can be:

- Accessible from the soft-core only,
- Accessible from the hardware only, or
- Shared memory that is accessible from both the software and the hardware.

All types of memory can be intern (on the FPGA) or extern to the FPGA.

Figure 3.1. FPGA with Soft-core, ASP and Memory



Data can be shared between the ASP and the soft-core in two ways: via the I/O bus or via shared memory.

1. ***Data sharing via I/O bus***

   - Data is passed as function parameters from a software function to a hardware function.
   - Data is passed as return value from a hardware function to the software caller.

2. ***Data sharing via shared memory***

   A data object that should be available to both the hardware and the software, could be allocated in shared memory that is both connected to the ASP as to the soft-core. Both the ASP and the soft-core need to know the address of the data object. This can be done in two ways:

   - Pass a pointer value to the data object as parameter to the hardware function. Make sure the data object is allocated in shared memory.
   - Use the `__at()` keyword to locate the data object at an absolute address. (See Section 3.5.4, *Placing a Data Object at an Absolute Address: __at()*.

3.6. Libraries

The main difference between traditional C libraries and the CHC C library, is the format in which the library is distributed. The CHC compiler's C library is a MIL archive. MIL, the *Medium Level Intermediate Language*, is a language used by TASKING compilers to represent the source code in a format that is suited for code generation by the compiler back-end.

The CHC C library was created by translating the source code of the C library into the MIL format. The MIL format is both an output and input format for the compiler. Subsequently, the MIL files were grouped together by the archiver into a library.

The C library also contains the startup code. The startup code is qualified `__export` and calls the `main()` function. At least one function in a program should be `__export` qualified, otherwise no hardware functions will be instantiated. The C library should be listed as one of the files that are passed to the compiler.

Example

```
chc file.c .../lib/libc.ma
```

Compile file `file.c` and link it with the C library `libc.ma`. If `file.c` contains a `main()` then the startup code is extracted form `libc.ma`.

Library	Description
libc.ma	32-bit C library (big endian)
libc16.ma	16-bit C library (big endian)
libcle.ma	32-bit C library (little endian)
libc16le.ma	16-bit C library (little endian)

Chapter 4. Using the CHC Compiler

This chapter explains the compilation process and how to invoke the compiler when building your project.

4.1. Invocation and Operating Modes

The CHC compiler is fully integrated in Altium Designer and operates always in concert with an embedded toolset.

Functions that are `__rtl` qualified, are processed by the CHC compiler and are translated into an electronic circuit, whereas the non `__rtl` qualified functions are processed by the embedded compiler and are translated into assembly code (see Section 3.4, *Function and Symbol Qualifiers*). The assembly code is then further assembled and linked using a traditional build flow.

Example: Compile to both software and hardware

This example shows how to create a system that runs on a processor core and uses a hardware function to off-load a computational intensive function to hardware.

jpeg.c

```
__export __rtl __CC(wishbone,0) void jpegdct(short *d, short *r);
void cleanup(void);

void main( void )
{
    ...
    jpegdct(short *d, short *r);
    ...
    cleanup();
}
```

jpeg_dct.c

```
__export __rtl __CC(wishbone,0) void jpegdct(short *d, short *r)
{
    ...
}

cleanup()
{
    ...
}
```

Invocation

To invoke the CHC compiler, two criteria must be met:

- The schematic of your project must contain an Application Specific Processor (ASP) component. The ASP component will contain the hardware compiled functions when your entire project has been built and synthesized.

- The ASP component needs to be configured to translate selected functions to hardware.

In the example above, functions are selected using the function qualifiers `__rtl`, `__export` and `__CC`. Instead of using these function qualifiers, you can also use the configuration dialog to mark these functions. To configure the ASP component for hardware compilation, proceed as follows:

1. On the schematic, right-click on the ASP component and select **Configure U*x* (WB_ASP) ...**

   The Configure Ux (WB_ASP Properties) dialog appears.

2. Enable both option **Generate ASP** and option **Use ASP from Software**.

   This will effectively invoke the CHC compiler when you build and synthesize your project. You can disable the option Use **ASP from software** to test your project without calling the hardware compiled functions. In this case all hardware will be generated, but because the embedded software compiler is now told to compile these functions to software, the software variant of the functions are called during execution. Note that this does not work if you typed the function qualifier `__rtl` in your source code by hand!

   Invoking the CHC compiler is only useful if functions are selected to translate to hardware. As explained in Section 3.4, *Function and Symbol Qualifiers*, you can either manually mark these functions in the C source with the function qualifier `__rtl`. However, you can also select them in this dialog:

3. In the **Symbols in Hardware** section, in the lower part under the **Implement in Hardware** column, select the functions that should be translated to hardware.

   If a hardware function is called by a software function, mark it as **Export to Software** too! (This is the equivalent of the `__export` function qualifier).

The **chc** compiler analyzes all C source files and creates an electronic circuit that implements the `jpegdct` function in this example. If file `jpeg.c` would not have been passed to the CHC compiler, the compiler would still be able to produce a correct hardware implementation for the `jpegdct` function. This implementation would probably be less efficient since the compiler is not able to apply application wide optimizations. For example, the compiler would not be able to detect whether the pointers `d` and `r` are aliases or not.

The **chc** compiler and the regular embedded compilers, such as the **c3000** compiler, understand the meaning of the `__export` `__rtl` and `__CC()` qualifiers. The **chc** compiler builds the necessary ”glue" logic to connect the generated electronic circuit to a TSK3000 processor core (interface via Wishbone bus). The **c3000** compiler calls the `jpegdct` function using a special calling convention that triggers the hardware function.

In Altium Designer, the build process is fully automated.

4.1.1. CHC Compiler Options *

Normally, you do not need to set compiler options; the fully automated build process uses default settings that work for all situations. Nevertheless, it is possible to set your own compiler options, for example to manually choose for certain optimizations. The CHC compiler options are not further explained in this manual, except where absolutely necessary. You can ask for a list of options with their descriptions by invoking the CHC compiler from the command line:

1. Make sure that you have started Altium Designer to gain licensed access to the CHC compiler on the command line. In Altium Designer, recompile your embedded project to activate the license.
2. Open a command prompt window and browse to `...\System\Tasking\chc\bin` in the installation directory of Altium Designer.
3. Invoke the CHC compiler with the command **chc -?o**. This gives a list of all available options with their descriptions.

To manually set a CHC compiler option in Altium Designer, you need to have an embedded project (which may be part of an FPGA project). To access the compiler options:

1. In the **Projects** panel, right-click on the name of the embedded project and select **Project Options...**

   The Options for Embedded Project .PrjEmb appears.
2. On the **Compiler Options** tab, in the left pane, expand the **C Compiler** entry and select **Miscellaneous**.
3. In the **Additional CHC compiler options** field, you can enter additional options for the CHC compiler.
4. Click **OK** to confirm the new settings and to close the dialog.

If you have an FPGA project with C sheet symbols but further no embedded project being part of the FPGA project, it is *not* possible to manually set additional CHC compiler options.

4.2. Simulating the Compiler Output

This section is only relevant if you are familiar with VHDL, VHDL simulation tools (for example ModelSim), and are interested in how the circuits that are generated by the **chc** compiler behave.

Please have a look at the file `...System\Tasking\chc\etc\pcls_driver.vhdl`.

This file is a test bench driver for compiler generated hardware circuits. The top level entity in file `pcls_driver.vhdl` is 'test_bench', it instantiates three components: `pcls_driver`, `c_root`, and `putchar`.

Component `c_root` is the top-level component created by the CHC compiler and contains the logic described in your C source files. Component `pcls_driver` activates the `c_root` component. First it resets and then activates component `c_root` by asserting the 'act' signal. The `pcls_driver` component also generates the clock signal. When the `c_root` component finishes, it asserts its 'done' signal. When this happens, the `pcls_driver` prints the number of clock cycles consumed by the `c_root` component. Component `putchar` implements the C library function `__putchar()`. This function prints characters using the facilities of the `textio` package in the VHDL `std` library.

Example

main.c

```
{
    printf("It's a strange world");
}
```

Invocation

```
chc main.c .../lib/libc.ma -omain.vhdl
```

When you load the files `pcls_driver.vhdl` and `main.vhdl` in your VHDL simulator and simulate the `test_bench` entity, you will see the text "It's a strange world" in the simulator's text output window.

See also the file `...\System\Tasking\chc\lib\src\cstart.c`. The file `cstart.c` contains the startup code that is executed before the main function is called.

The test driver expects that all memory devices are created by the compiler. If you specify (in the LSL file) that the compiler generated circuit interfaces with extern memory then the `pcls_driver` should be modified and provide an interface to the memory.

4.3. Synthesizing the Compiler Output

The generated RTL is formatted in accordance with design guidelines provided by Altera, Altium, Synplicity and Xilinx synthesis tools. FPGA device specific building blocks are automatically inferred from the compiler generated RTL by these synthesizers. Unless syntax requirements of the synthesizers conflict with the 'IEEE P1076.6 Standard For VHDL Register Transfer Level Synthesis', or the '1364.1 IEEE Standard for Verilog Register Transfer Level Synthesis', the generated RTL complies with the IEEE standards.

Extern resources, i.e. resources defined in the resource definition file with attribute `extern=1`, are not instantiated by the compiler and should be passed to the synthesis tool.

4.4. How the Compiler Searches Include Files

When you use include files (with the `#include` statement), you can specify their location in several ways. The compiler searches the specified locations in the following order:

1. If the `#include` statement contains an absolute path name, the compiler looks for the specified file in the specified directory. If no path is specified, or a relative path is specified, the compiler looks in the same directory as the location of the source file. This is only possible for include files that are enclosed in "".

   This first step is not done for include files enclosed in <>.

2. When the compiler did not find the include file, it looks in the directories that are specified with the option **Include files path**. (While in your C source file, from the **Project** menu select **Project options...** and go to the **Build Options**).

   You can set this option for the embedded project, but it is shared with the CHC compiler.

3. When the compiler did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks in the path(s) specified in the environment variable `CHCINC`.

4. When the compiler still did not find the include file, it finally tries the default include directory relative to the installation directory.

Example

Suppose that the C source file `test.c` contains the following lines:

```
#include <stdio.h>
#include "myinc.h"
```

First the compiler looks for the file `stdio.h` in the specified directory. Because no directory path is specified with `stdio.h`, the compiler searches in the environment variable `CHCINC` and then in the default `include` directory.

The compiler now looks for the file `myinc.h`, in the directory where `test.c` is located. If the file was not found, the compiler searches in the environment variable `CHCINC` and then in the default `include` directory.

4.5. How the Compiler Searches the C library

If your code calls functions defined in the C library you should pass the path to the C library at the command line. The compiler will link the function definitions with your code.

The 32-bit version of the C library is named `libc.ma`. The 16-bit version of the C library is named `libc16.ma`. Both libraries are located in directory `...\System\Tasking\chc\lib\`. The compiler does not search in other directories than the one specified at the command line, nor are there any options or environment variables available to specify a search path.

The C library is a MIL archive. It is created by compiling all C library source modules to the MIL format, the resulting MIL files are combined into one archive (i.e. library) from which the compiler extracts the required functions.

4.6. Rebuilding the C Library

All sources for the C library are shipped with the product. You can rebuild the C library by executing the makefiles located in directories `...\System\Tasking\chc\lib\src\libc` and `...\System\Tasking\chc\lib\src\libc16` to recreate the 32-bit, respectively the 16-bit versions of the C library.

The command to execute the make file is:

```
mkhc makefile
```

You may want to rebuild the library, for example to change the size of the heap which has a size of 200 bytes by default.

4.7. Debugging the Generated Code

The compiler generated VHDL or Verilog is *correct by construction*. This means that if the C code is free of bugs then the generated code is also free of bugs. The only way to debug a real electronic circuit is to analyze waveforms captured by either a HDL simulator or a (virtual) logic analyzer. Analyzing the waveforms is tedious and time consuming and correlating the waveforms to the C source is possible but difficult. So, before you compile a code fragment to hardware, you first have to be sure that the C source code is correct.

Before you compile your code to hardware:

- you should analyze and fix all warning messages issued by the compiler
- you may enable MISRA-C code checking and analyze and fix all warnings
- you should debug and run your software on an embedded processor core. Alternatively you can debug your code on a PC.

Finally you should instantiate the debugged functions in hardware.

4.8. C Code Checking: MISRA-C

The C programming language is a standard for high level language programming in embedded systems, yet it is considered somewhat unsuitable for programming safety-related applications. Through enhanced code checking and strict enforcement of best practice programming rules, TASKING MISRA-C code checking helps you to produce more robust code.

MISRA-C specifies a subset of the C programming language which is intended to be suitable for embedded automotive systems. It consists of a set of rules, defined in *MISRA-C:2004, Guidelines for the Use of the C Language in Critical Systems* (Motor Industry Research Association (MIRA), 2004). To enable MISRA-C checking:

1. Make the (or one of the) C source files visible in your workspace.
2. From the **Project** menu choose **Project Options...**

   The Options for Embedded Project dialog appears.
3. Expand the **C compiler** entry and select **MISRA-C**.
4. Set MISRA-C rules to **All supported MISRA-C rules**.

For a complete overview of all MISRA-C rules, see Chapter 6, *MISRA-C Rules*.

Implementation issues

The MISRA-C implementation in the compiler supports nearly all rules. Only a few rules are not supported because they address documentation, run-time behavior, or other issues that cannot be checked by static source code inspection.

During compilation of the code, violations of the enabled MISRA-C rules are indicated with error messages and the build process is halted.

MISRA-C rules are divided in required rules and advisory rules. If rules are violated, errors are generated causing the compiler to stop. With the following options warnings, instead of errors, are generated for either or both the required rules and the advisory rules. Being still in the MISRA-C pane:

- In the right pane, enable the options **Turn advisory rule violation into warning** and/or **Turn required rule violation into warning**.

Not all MISRA-C violations will be reported when other errors are detected in the input source. For instance, when there is a syntax error, all semantic checks will be skipped, including some of the MISRA-C checks. Also note that some checks cannot be performed when the optimizations are switched off.

4.9. C Compiler Error Messages

The C compiler reports the following types of error messages:

F (Fatal errors)

After a fatal error the compiler immediately aborts compilation.

E (Errors)

Errors are reported, but the compiler continues compilation. No output files are produced.

W (Warnings)

Warning messages do not result into an erroneous output file. They are meant to draw your attention to assumptions of the compiler for a situation which may not be correct. You can control warnings with embedded compiler option **Treat warnings as errors** which is passed to the CHC compiler as well:

1. While in your C source, from the Project menu select Project Options...

   The Options for Embedded Project dialog appears.

2. Expand the **C Compiler entry** and select **Diagnostics**.

3. Set **Treat warnings as errors** to **True**

I (Information)

Information messages are always preceded by an error message. Information messages give extra information about the error.

S (System errors)

System errors occur when internal consistency checks fail and should never occur. When you still receive the system error message

```
S9##: internal consistency check failed - please report
```

please report the error number and as many details as possible about the context in which the error occurred.

Chapter 5. Libraries

5.1. Introduction

This chapter contains an overview of all library functions that you can call in your C source. This includes all functions of the standard C library (`libc.ma`).

Section 5.2, *Library Functions*, gives an overview of all library functions you can use, grouped per header file. A number of functions declared in `wchar.h` are parallel to functions in other header files. These are discussed together.

Section 5.3, *C Library Reentrancy*, gives an overview of which functions are reentrant and which are not.

The following libraries are included in the **chc** toolset. Both Altium Designer and the control program **cchc** automatically select the appropriate libraries depending on the specified **chc** derivative.

Library	Description
libc.ma	32-bit C library (big endian)
libc16.ma	16-bit C library (big endian)
libcle.ma	32-bit C library (little endian)
libc16le.ma	16-bit C library (little endian)

5.2. Library Functions

The tables in the sections below list all library functions, grouped per header file in which they are declared. Some functions are not completely implemented because their implementation depends on the context where your application will run. These functions are for example all I/O related functions.

5.2.1. assert.h

`assert(`*`expr`*`)` Prints a diagnostic message if NDEBUG is not defined. (Implemented as macro)

5.2.2. complex.h

The current version of the CHC compiler does not support the type specifiers `_Complex` and `_Imaginary`. Therefore the functions in this include file are not supported.

The complex number *z* is also written as *x*+*y*i where *x* (the real part) and *y* (the imaginary part) are real numbers of types `float`, `double` or `long double`. The real and imaginary part can be stored in structs or in arrays. This implementation uses arrays because structs may have different alignments.

The header file `complex.h` also defines the following macros for backward compatibility:

```
complex    _Complex    /* C99 keyword */
imaginary  _Imaginary  /* C99 keyword */
```

Parallel sets of functions are defined for double, float and long double. They are respectively named *function*, *function*`f`, *function*`l`. All long type functions, though declared in `complex.h`, are implemented as the `double` type variant which nearly always meets the requirement in embedded applications.

This implementation uses the obvious implementation for complex multiplication; and a more sophisticated implementation for division and absolute value calculations which handles underflow, overflow and infinities with more care. The ISO C99 `#pragma CX_LIMITED_RANGE` therefore has no effect.

Trigonometric functions

`csin`	`csinf`	`csinl`	Returns the complex sine of *z*.
`ccos`	`ccosf`	`ccosl`	Returns the complex cosine of *z*.
`ctan`	`ctanf`	`ctanl`	Returns the complex tangent of *z*.
`casin`	`casinf`	`casinl`	Returns the complex arc sine $\sin^{-1}(z)$.
`cacos`	`cacosf`	`cacosl`	Returns the complex arc cosine $\cos^{-1}(z)$.
`catan`	`catanf`	`catanl`	Returns the complex arc tangent $\tan^{-1}(z)$.
`csinh`	`csinhf`	`csinhl`	Returns the complex hyperbolic sine of *z*.
`ccosh`	`ccoshf`	`ccoshl`	Returns the complex hyperbolic cosine of *z*.
`ctanh`	`ctanhf`	`ctanhl`	Returns the complex hyperbolic tangent of *z*.
`casinh`	`casinhf`	`casinhl`	Returns the complex arc hyperbolic sinus of *z*.
`cacosh`	`cacoshf`	`cacoshl`	Returns the complex arc hyperbolic cosine of *z*.
`catanh`	`catanhf`	`catanhl`	Returns the complex arc hyperbolic tangent of *z*.

Exponential and logarithmic functions

`cexp`	`cexpf`	`cexpl`	Returns the result of the complex exponential function e^z.
`clog`	`clogf`	`clogl`	Returns the complex natural logarithm.

Power and absolute-value functions

`cabs`	`cabsf`	`cabsl`	Returns the complex absolute value of *z* (also known as *norm*, *modulus* or *magnitude*).
`cpow`	`cpowf`	`cpowl`	Returns the complex value of *x* raised to the power *y* (x^y) where both *x* and *y* are complex numbers.
`csqrt`	`csqrtf`	`csqrtl`	Returns the complex square root of *z*.

Manipulation functions

`carg`	`cargf`	`cargl`	Returns the argument of z (also known as *phase angle)*.
`cimag`	`cimagf`	`cimagl`	Returns the imaginary part of *z* as a real (respectively as a `double`, `float`, `long double`)

`conj`	`conjf`	`conjl`	Returns the complex conjugate value (the sign of its imaginary part is reversed).
`cproj`	`cprojf`	`cprojl`	Returns the value of the projection of *z* onto the Riemann sphere.
`creal`	`crealf`	`creall`	Returns the real part of *z* as a real (respectively as a `double`, `float`, `long double`)

5.2.3. ctype.h and wctype.h

The header file `ctype.h` declares the following functions which take a character *c* as an integer type argument. The header file `wctype.h` declares parallel wide-character functions which take a character *c* of the `wchar_t` type as argument.

ctype.h	wctype.h	Description
`isalnum`	`iswalnum`	Returns a non-zero value when c is an alphabetic character or a number ([A-Z][a-z][0-9]).
`isalpha`	`iswalpha`	Returns a non-zero value when c is an alphabetic character ([A-Z][a-z]).
`isblank`	`iswblank`	Returns a non-zero value when c is a blank character (tab, space...)
`iscntrl`	`iswcntrl`	Returns a non-zero value when c is a control character.
`isdigit`	`iswditit`	Returns a non-zero value when c is a numeric character ([0-9]).
`isgraph`	`iswgraph`	Returns a non-zero value when c is printable, but not a space.
`islower`	`iswlower`	Returns a non-zero value when c is a lowercase character ([a-z]).
`isprint`	`iswprint`	Returns a non-zero value when c is printable, including spaces.
`ispunct`	`iswpunct`	Returns a non-zero value when c is a punctuation character (such as '.', ',', '!').
`isspace`	`iswspace`	Returns a non-zero value when c is a space type character (space, tab, vertical tab, formfeed, linefeed, carriage return).
`isupper`	`iswupper`	Returns a non-zero value when c is an uppercase character ([A-Z]).
`isxdigit`	`iswxdigit`	Returns a non-zero value when c is a hexadecimal digit ([0-9][A-F][a-f]).
`tolower`	`towlower`	Returns c converted to a lowercase character if it is an uppercase character, otherwise c is returned.
`toupper`	`towupper`	Returns c converted to an uppercase character if it is a lowercase character, otherwise c is returned.
`_tolower`	-	Converts c to a lowercase character, does not check if c really is an uppercase character. Implemented as macro. This macro function is not defined in ISO C99.
`_toupper`	-	Converts c to an uppercase character, does not check if c really is a lowercase character. Implemented as macro. This macro function is not defined in ISO C99.
`isascii`		Returns a non-zero value when c is in the range of 0 and 127. This function is not defined in ISO C99.

ctype.h	wctype.h	Description
toascii		Converts c to an ASCII value (strip highest bit). This function is not defined in ISO C99.

5.2.4. errno.h

`int errno` External variable that holds implementation defined error codes.

The following error codes are defined as macros in `errno.h`:

EPERM	1	Operation not permitted
ENOENT	2	No such file or directory
EINTR	3	Interrupted system call
EIO	4	I/O error
EBADF	5	Bad file number
EAGAIN	6	No more processes
ENOMEM	7	Not enough core
EACCES	8	Permission denied
EFAULT	9	Bad address
EEXIST	10	File exists
ENOTDIR	11	Not a directory
EISDIR	12	Is a directory
EINVAL	13	Invalid argument
ENFILE	14	File table overflow
EMFILE	15	Too many open files
ETXTBSY	16	Text file busy
ENOSPC	17	No space left on device
ESPIPE	18	Illegal seek
EROFS	19	Read-only file system
EPIPE	20	Broken pipe
ELOOP	21	Too many levels of symbolic links
ENAMETOOLONG	22	File name too long

Floating-point errors

EDOM	23	Argument too large
ERANGE	24	Result too large

Errors returned by printf/scanf

ERR_FORMAT	25	Illegal format string for printf/scanf
ERR_NOFLOAT	26	Floating-point not supported
ERR_NOLONG	27	Long not supported
ERR_NOPOINT	28	Pointers not supported

Encoding errors set by functions like fgetwc, getwc, mbrtowc, etc ...

EILSEQ	29	Invalid or incomplete multibyte or wide character

Errors returned by RTOS

ECANCELED	30	Operation canceled
ENODEV	31	No such device

5.2.5. fcntl.h

The header file `fcntl.h` contains the function `open()`, which calls the low level function `_open()`, and definitions of flags used by the low level function `_open()`. This header file is not defined in ISO C99.

`open`	Opens a file a file for reading or writing. Calls `_open`.

5.2.6. fenv.h

Contains mechanisms to control the floating-point environment. The functions in this header file are not implemented.

`fegetenv`	Stores the current floating-point environment. (*Not implemented*)
`feholdexept`	Saves the current floating-point environment and installs an environment that ignores all floating-point exceptions. (*Not implemented*)
`fesetenv`	Restores a previously saved (fegetenv or feholdexcept) floating-point environment. (*Not implemented*)
`feupdateenv`	Saves the currently raised floating-point exceptions, restores a previously saved floating-point environment and finally raises the saved exceptions. (*Not implemented*)
`feclearexcept`	Clears the current exception status flags corresponding to the flags specified in the argument. (*Not implemented*)
`fegetexceptflag`	Stores the current setting of the floating-point status flags. (*Not implemented*)
`feraiseexcept`	Raises the exceptions represented in the argument. As a result, other exceptions may be raised as well. (*Not implemented*)
`fesetexceptflag`	Sets the current floating-point status flags. (*Not implemented*)
`fetestexcept`	Returns the bitwise-OR of the exception macros corresponding to the exception flags which are currently set *and* are specified in the argument. (*Not implemented*)

For each supported exception, a macro is defined. The following exceptions are defined:

`FE_DIVBYZERO`	`FE_INEXACT`	`FE_INVALID`
`FE_OVERFLOW`	`FE_UNDERFLOW`	`FE_ALL_EXCEPT`

`fegetround`	Returns the current rounding direction, represented as one of the values of the rounding direction macros. (*Not implemented*)
`fesetround`	Sets the current rounding directions. (*Not implemented*)

Currently no rounding mode macros are implemented.

5.2.7. float.h

The header file `float.h` defines the characteristics of the real floating-point types `float`, `double` and `long double`.

`float.h` used to contain prototypes for the functions `copysign(f)`, `isinf(f)`, `isfinite(f)`, `isnan(f)` and `scalb(f)`. These functions have accordingly to the ISO C99 standard been moved to the header file `math.h`. See also Section 5.2.14, *math.h and tgmath.h*.

The following functions are only available for ISO C90:

`copysignf(float f,float s)`	Copies the sign of the second argument *s* to the value of the first argument *f* and returns the result.
`copysign(double d,double s)`	Copies the sign of the second argument *s* to the value of the first argument *d* and returns the result.
`isinff(float f)`	Test the variable *f* on being an infinite (IEEE-754) value.
`isinf(double d);`	Test the variable *d* on being an infinite (IEEE-754) value.
`isfinitef(float f)`	Test the variable *f* on being a finite (IEEE-754) value.
`isfinite(double d)`	Test the variable *d* on being a finite (IEEE-754) value.
`isnanf(float f)`	Test the variable *f* on being NaN (Not a Number, IEEE-754) .
`isnan(double d)`	Test the variable *d* on being NaN (Not a Number, IEEE-754) .
`scalbf(float f,int p)`	Returns *f* * 2^*p* for integral values without computing 2^N.
`scalb(double d,int p)`	Returns *d* * 2^*p* for integral values without computing 2^N. (See also `scalbn` in Section 5.2.14, *math.h and tgmath.h*)

5.2.8. inttypes.h and stdint.h

The header files `stdint.h` and `inttypes.h` provide additional declarations for integer types and have various characteristics. The `stdint.h` header file contains basic definitions of integer types of certain sizes, and corresponding sets of macros. This header file clearly refers to the corresponding sections in the ISO C99 standard.
The `inttypes.h` header file includes `stdint.h` and adds portable formatting and conversion functions. Below the conversion functions from `inttypes.h` are listed.

`imaxabs(intmax_t j)`	Returns the absolute value of j
`imaxdiv(intmax_t numer, intmax_t denom)`	Computes `numer/denom`and `numer % denom`. The result is stored in the `quot` and `rem` components of the `imaxdiv_t` structure type.
`strtoimax(const char * restrict nptr, char ** restrict endptr, int base)`	Convert string to maximum sized integer. (Compare `strtoll`)

`strtoumax(const char * restrict nptr, char ** restrict endptr, int base)`	Convert string to maximum sized unsigned integer. (Compare `strtoull`)
`wcstoimax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base)`	Convert wide string to maximum sized integer. (Compare `wcstoll`)
`wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base)`	Convert wide string to maximum sized unsigned integer. (Compare `wcstoull`)

5.2.9. io.h

The header file `io.h` contains prototypes for low level I/O functions. Definitions are located in the source file `pcls_io.c`. This header file is not defined in ISO C99.

`_close(`*`fd`*`)`	Used by the functions `close` and `fclose`.
`_lseek(`*`fd`*`,`*`offset`*`,`*`whence`*`)`	Used by all file positioning functions: `fgetpos`, `fseek`, `fsetpos`, `ftell`, `rewind`.
`_open(`*`fd`*`,`*`flags`*`)`	Used by the functions `fopen` and `freopen`.
`_read(`*`fd`*`,*`*`buff`*`,`*`cnt`*`)`	Reads a sequence of characters from a file.
`_unlink(*`*`name`*`)`	Used by the function remove.
`_write(`*`fd`*`,*`*`buffer`*`,`*`cnt`*`)`	Writes a sequence of characters to a file.

5.2.10. iso646.h

The header file `iso646.h` adds tokens that can be used instead of regular operator tokens.

```
#define and     &&
#define and_eq  &=
#define bitand  &
#define bitor   |
#define compl   ~
#define not     !
#define not_eq  !=
#define or      ||
#define or_eq   |=
#define xor     ^
#define xor_eq  ^=
```

5.2.11. limits.h

Contains the sizes of integral types, defined as macros.

5.2.12. locale.h

To keep C code reasonable portable across different languages and cultures, a number of facilities are provided in the header file `local.h`.

```
char *setlocale( int category, const char *locale )
```

The function above changes locale-specific features of the run-time library as specified by the category to change and the name of the locale.

The following categories are defined and can be used as input for this function:

```
LC_ALL      0      LC_NUMERIC   3
LC_COLLATE  1      LC_TIME      4
LC_CTYPE    2      LC_MONETARY  5
```

```
struct lconv *localeconv( void )
```

Returns a pointer to type `struct lconv` with values appropriate for the formatting of numeric quantities according to the rules of the current locale. The `struct lconv` in this header file is conforming the ISO standard.

5.2.13. malloc.h

The header file `malloc.h` contains prototypes for memory allocation functions. This include file is not defined in ISO C99, it is included for backwards compatibility with ISO C90. For ISO C99, the memory allocation functions are part of `stdlib.h`. See Section 5.2.22, *stdlib.h and wchar.h*.

`malloc(`*`size`*`)`	Allocates space for an object with size *size*. The allocated space is not initialized. Returns a pointer to the allocated space.
`calloc(`*`nobj`*`,`*`size`*`)`	Allocates space for n objects with size *size*. The allocated space is initialized with zeros. Returns a pointer to the allocated space.
`free(*`*`ptr`*`)`	Deallocates the memory space pointed to by *ptr* which should be a pointer earlier returned by the `malloc` or `calloc` function.
`realloc(*`*`ptr`*`,`*`size`*`)`	Deallocates the old object pointed to by *ptr* and returns a pointer to a new object with size *size*. The new object cannot have a size larger than the previous object.

5.2.14. math.h and tgmath.h

The header file `math.h` contains the prototypes for many mathematical functions. Before ISO C99, all functions were computed using the double type (the float was automatically converted to double, prior to calculation). In this ISO C99 version, parallel sets of functions are defined for `double`, `float` and `long double`. They are respectively named *function*, *function*`f`, *function*`l`. All `long` type functions, though declared in `math.h`, are implemented as the `double` type variant which nearly always meets the requirement in embedded applications.

The header file `tgmath.h` contains parallel type generic math macros whose expansion depends on the used type. `tgmath.h` includes `math.h` and the effect of expansion is that the correct `math.h` functions are called. The type generic macro, if available, is listed in the second column of the tables below.

Trigonometric and hyperbolic functions

math.h			tgmath.h	Description
`sin`	`sinf`	`sinl`	`sin`	Returns the sine of `x`.
`cos`	`cosf`	`cosl`	`cos`	Returns the cosine of `x`.
`tan`	`tanf`	`tanl`	`tan`	Returns the tangent of `x`.
`asin`	`asinf`	`asinl`	`asin`	Returns the arc sine $\sin^{-1}(x)$ of `x`.
`acos`	`acosf`	`acosl`	`acos`	Returns the arc cosine $\cos^{-1}(x)$ of `x`.
`atan`	`atanf`	`atanl`	`atan`	Returns the arc tangent $\tan^{-1}(x)$ of `x`.
`atan2`	`atan2f`	`atan2l`	`atan2`	Returns the result of: $\tan^{-1}(y/x)$.
`sinh`	`sinhf`	`sinhl`	`sinh`	Returns the hyperbolic sine of `x`.
`cosh`	`coshf`	`coshl`	`cosh`	Returns the hyperbolic cosine of `x`.
`tanh`	`tanhf`	`tanhl`	`tanh`	Returns the hyperbolic tangent of `x`.
`asinh`	`asinhf`	`asinhl`	`asinh`	Returns the arc hyperbolic sine of `x`.
`acosh`	`acoshf`	`acoshl`	`acosh`	Returns the non-negative arc hyperbolic cosine of `x`.
`atanh`	`atanhf`	`atanhl`	`atanh`	Returns the arc hyperbolic tangent of `x`.

Exponential and logarithmic functions

All of these functions are new in ISO C99, except for `exp`, `log` and `log10`.

math.h			tgmath.h	Description
`exp`	`expf`	`expl`	`exp`	Returns the result of the exponential function e^x.
`exp2`	`exp2f`	`exp2l`	`exp2`	Returns the result of the exponential function 2^x. (*Not implemented*)
`expm1`	`expm1f`	`expm1l`	`expm1`	Returns the result of the exponential function e^x-1. (*Not implemented*)
`log`	`logf`	`logl`	`log`	Returns the natural logarithm `ln(x), x>0`.
`log10`	`log10f`	`log10l`	`log10`	Returns the base-10 logarithm of `x, x>0`.
`log1p`	`log1pf`	`log1pl`	`log1p`	Returns the base-e logarithm of `(1+x). x <> -1`. (*Not implemented*)
`log2`	`log2f`	`log2l`	`log2`	Returns the base-2 logarithm of `x. x>0`. (*Not implemented*)
`ilogb`	`ilogbf`	`ilogbl`	`ilogb`	Returns the signed exponent of x as an integer. `x>0`. (*Not implemented*)
`logb`	`logbf`	`logbl`	`logb`	Returns the exponent of `x` as a signed integer in value in floating-point notation. `x > 0`. (*Not implemented*)

frexp, ldexp, modf, scalbn, scalbln

math.h			tgmath.h	Description
frexp	frexpf	frexpl	frexp	Splits a float x into fraction *f* and exponent *n*, so that: *f* = 0.0 or 0.5 ≤ | *f* | ≤ 1.0 and *f**2^n = x. Returns *f*, stores *n*.
ldexp	ldexpf	ldexpl	ldexp	Inverse of frexp. Returns the result of x*2^n. (x and n are both arguments).
modf	modff	modfl	-	Splits a float x into fraction *f* and integer *n*, so that: | *f* | < 1.0 and *f*+*n*=x. Returns *f*, stores *n*.
scalbn	scalbnf	scalbnl	scalbn	Computes the result of x*FLT_RADIX^n. efficiently, not normally by computing FLT_RADIX^n explicitly.
scalbln	scalblnf	scalblnl	scalbln	Same as scalbn but with argument n as long int.

Rounding functions

math.h			tgmath.h	Description
ceil	ceilf	ceill	ceil	Returns the smallest integer not less than x, as a double.
floor	floorf	floorl	floor	Returns the largest integer not greater than x, as a double.
rint	rintf	rintl	rint	Returns the rounded integer value as an int according to the current rounding direction. See fenv.h. (*Not implemented*)
lrint	lrintf	lrintl	lrint	Returns the rounded integer value as a long int according to the current rounding direction. See fenv.h. (*Not implemented*)
llrint	lrintf	lrintl	llrint	Returns the rounded integer value as a long long int according to the current rounding direction. See fenv.h. (*Not implemented*)
nearbyint	nearbyintf	nearbyintl	nearbyint	Returns the rounded integer value as a floating-point according to the current rounding direction. See fenv.h. (*Not implemented*)
round	roundf	roundl	round	Returns the nearest integer value of x as int. (*Not implemented*)
lround	lroundf	lroundl	lround	Returns the nearest integer value of x as long int. (*Not implemented*)
llround	lroundf	llroundl	llround	Returns the nearest integer value of x as long long int. (*Not implemented*)
trunc	truncf	truncl	trunc	Returns the truncated integer value x. (*Not implemented*)

Remainder after division

math.h			tgmath.h	Description
fmod	fmodf	fmodl	fmod	Returns the remainder r of x-ny. n is chosen as trunc(*x/y*). r has the same sign as x.

math.h			tgmath.h	Description
`remainder`	`remainderf`	`remainderl`	`remainder`	Returns the remainder `r` of `x-ny`. `n` is chosen as `trunc(x/y)`. `r` may not have the same sign as `x`. (*Not implemented*)
`remquo`	`remquof`	`remquol`	`remquo`	Same as remainder. In addition, the argument `*quo` is given a specific value (see ISO). (*Not implemented*)

Power and absolute-value functions

math.h			tgmath.h	Description
`cbrt`	`cbrtf`	`cbrtl`	`cbrt`	Returns the real cube root of x ($=x^{1/3}$). (*Not implemented*)
`fabs`	`fabsf`	`fabsl`	`fabs`	Returns the absolute value of x (`|x|`). (`abs`, `labs`, `llabs`, `div`, `ldiv`, `lldiv` are defined in `stdlib.h`)
`fma`	`fmaf`	`fmal`	`fma`	Floating-point multiply add. Returns `x*y+z`. (*Not implemented*)
`hypot`	`hypotf`	`hypotl`	`hypot`	Returns the square root of x^2+y^2.
`pow`	`powf`	`powl`	`power`	Returns x raised to the power y (x^y).
`sqrt`	`sqrtf`	`sqrtl`	`sqrt`	Returns the non-negative square root of `x`. `x 0`.

Manipulation functions: copysign, nan, nextafter, nexttoward

math.h			tgmath.h	Description
`copysign`	`copysignf`	`copysignll`	`copysign`	Returns the value of `x` with the sign of `y`.
`nan`	`nanf`	`nanl`	`-`	Returns a quiet NaN, if available, with content indicated through *`tagp`*. (*Not implemented*)
`nextafter`	`nextafterf`	`nextafterl`	`nextafter`	Returns the next representable value in the specified format after `x` in the direction of `y`. Returns `y` is `x=y`. (*Not implemented*)
`nexttoward`	`nexttowardf`	`nexttowardl`	`nexttoward`	Same as `nextafter`, except that the second argument in all three variants is of type long double. Returns `y` if `x=y`. (*Not implemented*)

Positive difference, maximum, minimum

math.h			tgmath.h	Description
`fdim`	`fdimf`	`fdiml`	`fdim`	Returns the positive difference between: `|x-y|`. (*Not implemented*)
`fmax`	`fmaxf`	`fmaxl`	`fmax`	Returns the maximum value of their arguments. (*Not implemented*)
`fmin`	`fminf`	`fminl`	`fmin`	Returns the minimum value of their arguments. (*Not implemented*)

Error and gamma (Not implemented)

math.h			tgmath.h	Description
`erf`	`erff`	`erfl`	`erf`	Computes the error function of x. (*Not implemented*)
`erfc`	`erfcf`	`erfcl`	`erc`	Computes the complementary error function of x. (*Not implemented*)
`lgamma`	`lgammaf`	`lgammal`	`lgamma`	Computes the $*\log_e\lvert\Gamma(x)\rvert$ (*Not implemented*)
`tgamma`	`tgammaf`	`tgammal`	`tgamma`	Computes $\Gamma(x)$ (*Not implemented*)

Comparison macros

The next are implemented as macros. For any ordered pair of numeric values exactly one of the relationships - *less*, *greater*, and *equal* - is true. These macros are type generic and therefor do not have a parallel function in `tgmath.h`. All arguments must be expressions of real-floating type.

math.h	tgmath.h	Description
`isgreater`	-	Returns the value of `(x) > (y)`
`isgreaterequal`	-	Returns the value of `(x) >= (y)`
`isless`	-	Returns the value of `(x) < (y)`
`islessequal`	-	Returns the value of `(x) <= (y)`
`islessgreater`	-	Returns the value of `(x) < (y) || (x) > (y)`
`isunordered`	-	Returns 1 if its arguments are unordered, 0 otherwise.

Classification macros

The next are implemented as macros. These macros are type generic and therefor do not have a parallel function in `tgmath.h`. All arguments must be expressions of real-floating type.

math.h	tgmath.h	Description
`fpclassify`	-	Returns the class of its argument: `FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL` or `FP_ZERO`
`isfinite`	-	Returns a nonzero value if and only if its argument has a finite value
`isinf`	-	Returns a nonzero value if and only if its argument has an infinite value
`isnan`	-	Returns a nonzero value if and only if its argument has NaN value.
`isnormal`	-	Returns a nonzero value if an only if its argument has a normal value.

math.h	tgmath.h	Description
`signbit`	-	Returns a nonzero value if and only if its argument value is negative.

5.2.15. setjmp.h

The current version of the CHC compiler does not support the functions `setjmp()` and `longjmp()`.

The `setjmp` and `longjmp` in this header file implement a primitive form of non-local jumps, which may be used to handle exceptional situations. This facility is traditionally considered more portable than `signal.h`

`int setjmp(jmp_buf env)`	Records its caller's environment in `env` and returns 0.
`void longjmp(jmp_buf env, int status)`	Restores the environment previously saved with a call to `setjmp()`.

5.2.16. signal.h

Signals are possible asynchronous events that may require special processing. Each signal is named by a number. The following signals are defined:

`SIGINT`	1	Receipt of an interactive attention signal
`SIGILL`	2	Detection of an invalid function message
`SIGFPE`	3	An erroneous arithmetic operation (for example, zero divide, `overflow`)
`SIGSEGV`	4	An invalid access to storage
`SIGTERM`	5	A termination request sent to the program
`SIGABRT`	6	Abnormal termination, such as is initiated by the `abort` function

The next function sends the signal *sig* to the program:

```
int raise(int sig)
```

The next function determines how subsequent signals will be handled:

```
signalfunction *signal (int, signalfunction *);
```

The first argument specifies the signal, the second argument points to the signal-handler function or has one of the following values:

`SIG_DFL`	Default behavior is used
`SIG_IGN`	The signal is ignored

The function returns the previous value of `signalfunction` for the specific signal, or `SIG_ERR` if an error occurs.

5.2.17. stdarg.h

The facilities in this header file gives you a portable way to access variable arguments lists, such as needed for as `fprintf` and `vfprintf`. `va_copy` is new in ISO C99. This header file contains the following macros:

`va_arg(va_list ap,type)`	Returns the value of the next argument in the variable argument list. It's return type has the type of the given argument `type`. A next call to this macro will return the value of the next argument.
`va_copy(va_list dest, va_list src)`	This macro duplicates the current state of `src` in `dest`, creating a second pointer into the argument list. After this call, va_arg() may be used on `src` and `dest` independently.
`va_end(va_list ap)`	This macro must be called after the arguments have been processed. It should be called before the function using the macro 'va_start' is terminated.
`va_start(va_list ap, lastarg)`	This macro initializes `ap`. After this call, each call to va_arg() will return the value of the next argument. In our implementation, `va_list` cannot contain any bit type variables. Also the given argument `lastarg` must be the last non-bit type argument in the list.

5.2.18. stdbool.h

This header file contains the following macro definitions. These names for boolean type and values are consistent with C++. You are allowed to `#undefine` or redefine the macros below.

```
#define bool                              _Bool
#define true                              1
#define false                             0
#define __bool_true_false_are_defined     1
```

5.2.19. stddef.h

This header file defines the types for common use:

`ptrdiff_t`	Signed integer type of the result of subtracting two pointers.
`size_t`	Unsigned integral type of the result of the `sizeof` operator.
`wchar_t`	Integer type to represent character codes in large character sets.

Besides these types, the following macros are defined:

`NULL`	Expands to 0 (zero).
`offsetof(_type, _member)`	Expands to an integer constant expression with type `size_t` that is the offset in bytes of `_member` within structure type `_type`.

5.2.20. stdint.h

See Section 5.2.8, *inttypes.h and stdint.h*

5.2.21. stdio.h and wchar.h

Types

The header file `stdio.h` contains functions for performing input and output. A number of functions also have a parallel wide character function or macro, defined in `wchar.h`. The header file `wchar.h` also includes `stdio.h`.

In the C language, many I/O facilities are based on the concept of streams. The `stdio.h` header file defines the data type **`FILE`** which holds the information about a stream. A `FILE` object is created with the function `fopen`. The pointer to this object is used as an argument in many of the in this header file. The `FILE` object can contain the following information:

- the current position within the stream
- pointers to any associated buffers
- indications of for read/write errors
- end of file indication

The header file also defines type `fpos_t` as an `unsigned long`.

Macros

stdio.h	Description
`NULL`	Expands to 0 (zero).
`BUFSIZ`	Size of the buffer used by the `setbuf`/`setvbuf` function: 512
`EOF`	End of file indicator. Expands to -1.
`WEOF`	End of file indicator. Expands to UINT_MAX (defined in `limits.h`) NOTE: WEOF need not to be a negative number as long as its value does not correspond to a member of the wide character set. (Defined in `wchar.h`).
`FOPEN_MAX`	Number of files that can be opened simultaneously: 10
`FILENAME_MAX`	Maximum length of a filename: 100

`_IOFBF`
`_IOLBF`
`_IONBF`
	Expand to an integer expression, suitable for use as argument to the `setvbuf` function.

`L_tmpnam`	Size of the string used to hold temporary file names: 8 (tmp*xxxxx*)
`TMP_MAX`	Maximum number of unique temporary filenames that can be generated: 0x8000

`SEEK_CUR`
`SEEK_END`
`SEEK_SET`
	Expand to an integer expression, suitable for use as the third argument to the `fseek` function.

`stderr`
`stdin`
`stdout`
	Expressions of type "pointer to FILE" that point to the FILE objects associated with standard error, input and output streams.

File access

stdio.h	Description
`fopen(`*`name,mode`*`)`	Opens a file for a given mode. Available modes are: `"r"` read; open text file for reading `"w"` write; create text file for writing; if the file already exists, its contents is discarded `"a"` append; open existing text file or create new text file for writing at end of file `"r+"` open text file for update; reading and writing `"w+"` create text file for update; previous contents if any is discarded `"a+"` append; open or create text file for update, writes at end of file
`fclose(`*`name`*`)`	Flushes the data stream and closes the specified file that was previously opened with fopen.
`fflush(`*`name`*`)`	If stream is an output stream, any buffered but unwritten date is written. Else, the effect is undefined.
`freopen(`*`name,mode, stream`*`)`	Similar to fopen, but rather than generating a new value of type FILE *, the existing value is associated with a new stream.
`setbuf(`*`stream,buffer`*`)`	If buffer is NULL, buffering is turned off for the stream. Otherwise, setbuf is equivalent to: `(void) setvbuf(`*`stream,buffer`*`,_IOFBF,BUFSIZ)`.
setvbuf(*`stream,buffer,mode, size`*`)`	Controls buffering for the *stream*; this function must be called before reading or writing. *Mode* can have the following values: `_IOFBF` causes full buffering `_IOLBF` causes line buffering of text files `_IONBF` causes no buffering. If *buffer* is not NULL, it will be used as a buffer; otherwise a buffer will be allocated. *size* determines the buffer size.

Formatted input/output

The `format` string of **`printf`** related functions can contain plain text mixed with conversion specifiers. Each conversion specifier should be preceded by a '%' character. The conversion specifier should be built in order:

- Flags (in any order):

  `-` specifies left adjustment of the converted argument.

  `+` a number is always preceded with a sign character.
  + has higher precedence than `space`.

  space a negative number is preceded with a sign, positive numbers with a space.

  `0` specifies padding to the field width with zeros (only for numbers).

#	specifies an alternate output form. For o, the first digit will be zero. For x or X, "0x" and "0X" will be prefixed to the number. For e, E, f, g, G, the output always contains a decimal point, trailing zeros are not removed.

- A number specifying a minimum field width. The converted argument is printed in a field with at least the length specified here. If the converted argument has fewer characters than specified, it will be padded at the left side (or at the right when the flag '`-`' was specified) with spaces. Padding to numeric fields will be done with zeros when the flag '`0`' is also specified (only when padding left). Instead of a numeric value, also '`*`' may be specified, the value is then taken from the next argument, which is assumed to be of type `int`.
- A period. This separates the minimum field width from the precision.
- A number specifying the maximum length of a string to be printed. Or the number of digits printed after the decimal point (only for floating-point conversions). Or the minimum number of digits to be printed for an integer conversion. Instead of a numeric value, also '`*`' may be specified, the value is then taken from the next argument, which is assumed to be of type `int`.
- A length modifier 'h', 'hh', 'l', 'll', 'L', 'j', 'z' or 't'. 'h' indicates that the argument is to be treated as a `short` or `unsigned short`. 'hh' indicates that the argument is to be treated as a `char` or `unsigned char`. 'l' should be used if the argument is a `long` integer, 'll' for a `long long`. 'L' indicates that the argument is a `long double`. 'j' indicates a pointer to `intmax_t` or `uintmax_t`, 'z' indicates a pointer to `size_t` and 't' indicates a pointer to `ptrdiff_t`.

Flags, length specifier, period, precision and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined:

Character	Printed as
d, i	int, signed decimal
o	int, unsigned octal
x, X	int, unsigned hexadecimal in lowercase or uppercase respectively
u	int, unsigned decimal
c	int, single character (converted to unsigned char)
s	char *, the characters from the string are printed until a NULL character is found. When the given precision is met before, printing will also stop
f	double
e, E	double
g, G	double
a, A	double
n	int *, the number of characters written so far is written into the argument. This should be a pointer to an integer in default memory. No value is printed.
p	pointer
%	No argument is converted, a '%' is printed.

printf conversion characters

All arguments to the **`scanf`** related functions should be pointers to variables (in default memory) of the type which is specified in the format string.

The format string can contain :

- Blanks or tabs, which are skipped.
- Normal characters (not '%'), which should be matched exactly in the input stream.
- Conversion specifications, starting with a '%' character.

Conversion specifications should be built as follows (in order) :

- A '*', meaning that no assignment is done for this field.
- A number specifying the maximum field width.
- The conversion characters `d`, `i`, `n`, `o`, `u` and `x` may be preceded by 'h' if the argument is a pointer to `short` rather than `int`, or by 'hh' if the argument is a pointer to `char`, or by 'l' (letter ell) if the argument is a pointer to `long` or by 'll' for a pointer to `long long`, 'j' for a pointer to `intmax_t` or `uintmax_t`, 'z' for a pointer to `size_t` or 't' for a pointer to `ptrdiff_t`. The conversion characters `e`, `f`, and `g` may be preceded by 'l' if the argument is a pointer to `double` rather than `float`, and by 'L' for a pointer to a `long double`.
- A conversion specifier. '*', maximum field width and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Length specifier and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Character	Scanned as
d	int, signed decimal.
i	int, the integer may be given octal (i.e. a leading 0 is entered) or hexadecimal (leading "0x" or "0X"), or just decimal.
o	int, unsigned octal.
u	int, unsigned decimal.
x	int, unsigned hexadecimal in lowercase or uppercase.
c	single character (converted to unsigned char).
s	char *, a string of non white space characters. The argument should point to an array of characters, large enough to hold the string and a terminating NULL character.
f, F	float
e, E	float
g, G	float
a, A	float

Character	Scanned as
n	int *, the number of characters written so far is written into the argument. No scanning is done.
p	pointer; hexadecimal value which must be entered without 0x- prefix.
[...]	Matches a string of input characters from the set between the brackets. A NULL character is added to terminate the string. Specifying [ ]...] includes the ']' character in the set of scanning characters.
[^...]	Matches a string of input characters not in the set between the brackets. A NULL character is added to terminate the string. Specifying [^]...] includes the ']' character in the set.
%	Literal '%', no assignment is done.

scanf conversion characters

stdio.h	wchar.h	Description
`fscanf(stream, format, ...)`	`fwscanf(stream, format, ...)`	Performs a formatted read from the given *stream*. Returns the number of items converted successfully.
`scanf(format,...)`	`wscanf(format, ...)`	Performs a formatted read from `stdin`. Returns the number of items converted successfully.
`sscanf(*s, format, ...)`	`swscanf(*s, format, ...)`	Performs a formatted read from the string *s*. Returns the number of items converted successfully.
`vfscanf(stream, format, arg)`	`vfwscanf(stream, format, arg)`	Same as `fscanf`/`fwscanf`, but extra arguments are given as variable argument list *arg*. (See Section 5.2.17, *stdarg.h*)
`vscanf(format, arg)`	`vwscanf(format, arg)`	Same as `sscanf`/`swscanf`, but extra arguments are given as variable argument list *arg*. (See Section 5.2.17, *stdarg.h*)
`vsscanf(*s, format, arg)`	`vswscanf(*s, format, arg)`	Same as `scanf`/`wscanf`, but extra arguments are given as variable argument list *arg*. (See Section 5.2.17, *stdarg.h*)
`fprintf(stream, format, ...)`	`fwprintf(stream, format, ...)`	Performs a formatted write to the given *stream*. Returns EOF/WEOF on error.
`printf(format, ...)`	`wprintf(format, ...)`	Performs a formatted write to the stream `stdout`. Returns EOF/WEOF on error.
`sprintf(*s, format, ...)`	-	Performs a formatted write to string *s*. Returns EOF/WEOF on error.
`snprintf(*s, n, format, ...)`	`swprintf(*s, n, format, ...)`	Same as `sprintf`, but n specifies the maximum number of characters (including the terminating null character) to be written.
`vfprintf(stream, format, arg)`	`vfwprintf(stream, format, arg)`	Same as `fprintf`/`fwprintf`, but extra arguments are given as variable argument list *arg*. (See Section 5.2.17, *stdarg.h*)

stdio.h	wchar.h	Description
vprintf(*format*, *arg*)	vwprintf(*format*, *arg*)	Same as printf/wprintf, but extra arguments are given as variable argument list *arg*. (See Section 5.2.17, *stdarg.h*)
vsprintf(**s*, *format*, *arg*)	vswprintf(**s*, *format*, *arg*)	Same as sprintf/swprintf, but extra arguments are given as variable argument list *arg*. (See Section 5.2.17, *stdarg.h*)

Character input/output

stdio.h	wchar.h	Description
fgetc(*stream*)	fgetwc(*stream*)	Reads one character from *stream*. Returns the read character, or EOF/WEOF on error.
getc(*stream*)	getwc(*stream*)	Same as fgetc/fgetwc except that is implemented as a macro. NOTE: Currently #defined as getchar()/getwchar() because FILE I/O is not supported. Returns the read character, or EOF/WEOF on error.
getchar(stdin)	getwchar(stdin)	Reads one character from the stdin stream. Returns the character read or EOF/WEOF on error. Implemented as macro.
fgets(**s*, *n*, *stream*)	fgetws(**s*, *n*, *stream*)	Reads at most the next *n*-1 characters from the *stream* into array *s* until a newline is found. Returns s or NULL or EOF/WEOF on error.
gets(**s*, *n*, stdin)	-	Reads at most the next *n*-1 characters from the stdin stream into array *s*. A newline is ignored. Returns *s* or NULL or EOF/WEOF on error.
ungetc(*c*, *stream*)	ungetwc(*c*, *stream*)	Pushes character *c* back onto the input *stream*. Returns EOF/WEOF on error.
fputc(*c*, *stream*)	fputwc(*c*, *stream*)	Put character *c* onto the given *stream*. Returns EOF/WEOF on error.
putc(*c*, *stream*)	putwc(*c*, *stream*)	Same as fpuc/fputwc except that is implemented as a macro.
putchar(*c*, stdout)	putwchar(*c*, stdout)	Put character *c* onto the stdout stream. Returns EOF/WEOF on error. Implemented as macro.
fputs(**s*, *stream*)	fputws(**s*, *stream*)	Writes string *s* to the given *stream*. Returns EOF/WEOF on error.
puts(**s*)	-	Writes string *s* to the stdout stream. Returns EOF/WEOF on error.

Direct input/output

stdio.h	Description
`fread(ptr,size,nobj,stream)`	Reads *nobj* members of *size* bytes from the given *stream* into the array pointed to by *ptr*. Returns the number of elements successfully read.
`fwrite(ptr,size,nobj,stream)`	Writes *nobj* members of *size* bytes from to the array pointed to by *ptr* to the given *stream*. Returns the number of elements successfully written.

Random access

stdio.h	Description
`fseek(`*`stream`*`, `*`offset`*`, `*`origin`*`)`	Sets the position indicator for *stream*.

When repositioning a binary file, the new position *origin* is given by the following macros:

`SEEK_SET 0` *offset* characters from the beginning of the file
`SEEK_CUR 1` *offset* characters from the current position in the file
`SEEK_END 2` *offset* characters from the end of the file

`ftell(`*`stream`*`)`	Returns the current file position for *stream*, or -1L on error.

`rewind(`*`stream`*`)`	Sets the file position indicator for the *stream* to the beginning of the file. This function is equivalent to:
	`(void) fseek(stream,0L,SEEK_SET);`
	`clearerr(stream);`

`fgetpos(`*`stream`*`,`*`pos`*`)`	Stores the current value of the file position indicator for *stream* in the object pointed to by *pos*.
`fsetpos(`*`stream`*`,`*`pos`*`)`	Positions *`stream`* at the position recorded by `fgetpos` in **pos*.

Operations on files

stdio.h	Description
`remove(`*`file`*`)`	Removes the named file, so that a subsequent attempt to open it fails. Returns a non-zero value if not successful.
`rename(`*`old`*`,`*`new`*`)`	Changes the name of the file from old name to new name. Returns a non-zero value if not successful.
`tmpfile()`	Creates a temporary file of the mode "wb+" that will be automatically removed when closed or when the program terminates normally. Returns a `file` pointer.
`tmpnam(`*`buffer`*`)`	Creates new file names that do not conflict with other file names currently in use. The new file name is stored in a *buffer* which must have room for L_tmpnam characters. Returns a pointer to the temporary name. The file names are created in the current directory and all start with "tmp". At most TMP_MAX unique file names can be generated.

Error handling

stdio.h	Description
`clearerr(stream)`	Clears the end of file and error indicators for stream.
`ferror(stream)`	Returns a non-zero value if the error indicator for stream is set.
`feof(stream)`	Returns a non-zero value if the end of file indicator for stream is set.
`perror(*s)`	Prints *s* and the error message belonging to the integer `errno`. (See Section 5.2.4, *errno.h*)

5.2.22. stdlib.h and wchar.h

The header file `stdlib.h` contains general utility functions which fall into the following categories (Some have parallel wide-character, declared in `wchar.h`)

- Numeric conversions
- Random number generation
- Memory management
- Environment communication
- Searching and sorting
- Integer arithmetic
- Multibyte/wide character and string conversions.

Macros

`EXIT_SUCCES 0` `EXIT_FAILURE 1`	Predefined exit codes that can be used in the `exit` function.
`RAND_MAX 32767`	Highest number that can be returned by the `rand`/`srand` function.
`MB_CUR_MAX 1`	Maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category LC_CTYPE, see Section 5.2.12, *locale.h*).

Numeric conversions

The following functions convert the initial portion of a string **s* to a `double`, `int`, `long int` and `long long int` value respectively.

```
double      atof(*s)
int         atoi(*s)
long        atol(*s)
long long   atoll(*s)
```

The following functions convert the initial portion of the string **s* to a float, double and long double value respectively. **endp* will point to the first character not used by the conversion.

stdlib.h		wchar.h	
`float`	`strtof(*s,**endp)`	`float`	`wcstof(*s,**endp)`
`double`	`strtod(*s,**endp)`	`double`	`wcstod(*s,**endp)`
`long double`	`strtold(*s,**endp)`	`long double`	`wcstold(*s,**endp)`

The following functions convert the initial portion of the string **s* to a `long`, `long long`, `unsigned long` and `unsigned long long` respectively. Base specifies the radix. **endp* will point to the first character not used by the conversion.

stdlib.h	wchar.h
`long strtol (*s,**endp,base)`	`long wcstol (*s,**endp,base)`
`long long strtoll (*s,**endp,base)`	`long long wcstoll (*s,**endp,base)`
`unsigned long strtoul (*s,**endp,base)`	`unsigned long wcstoul (*s,**endp,base)`
`unsigned long long strtoull (*s,**endp,base)`	`unsigned long long wcstoull (*s,**endp,base)`

Random number generation

`rand`	Returns a pseudo random integer in the range 0 to RAND_MAX.
`srand(`*`seed`*`)`	Same as rand but uses *seed* for a new sequence of pseudo random numbers.

Memory management

`malloc(`*`size`*`)`	Allocates space for an object with size *size*. The allocated space is not initialized. Returns a pointer to the allocated space.
`calloc(`*`nobj`*`,`*`size`*`)`	Allocates space for n objects with size *size*. The allocated space is initialized with zeros. Returns a pointer to the allocated space.
`free(*`*`ptr`*`)`	Deallocates the memory space pointed to by *ptr* which should be a pointer earlier returned by the `malloc` or `calloc` function.
`realloc(*`*`ptr`*`,`*`size`*`)`	Deallocates the old object pointed to by *ptr* and returns a pointer to a new object with size *size*. The new object cannot have a size larger than the previous object.

Environment communication

`abort()`	Causes abnormal program termination. If the signal SIGABRT is caught, the signal handler may take over control. (See Section 5.2.16, *signal.h*).
`atexit(*`*`func`*`)`	*func* points to a function that is called (without arguments) when the program normally terminates.

`exit(status)`	Causes normal program termination. Acts as if `main()` returns with status as the return value. Status can also be specified with the predefined macros EXIT_SUCCES or EXIT_FAILURE.
`_Exit(status)`	Same as `exit`, but not registered by the `atexit` function or signal handlers registered by the `signal` function are called.
`getenv(*s)`	Searches an environment list for a string *s*. Returns a pointer to the contents of *s*. NOTE: this function is not implemented because there is no OS.
`system(*s)`	Passes the string *s* to the environment for execution. NOTE: this function is not implemented because there is no OS.

Searching and sorting

`bsearch(*key, *base, n, size, *cmp)`	This function searches in an array of *n* members, for the object pointed to by *key*. The initial base of the array is given by *base*. The size of each member is specified by *size*. The given array must be sorted in ascending order, according to the results of the function pointed to by *cmp*. Returns a pointer to the matching member in the array, or NULL when not found.
`qsort(*base, n, size, *cmp)`	This function sorts an array of *n* members using the quick sort algorithm. The initial base of the array is given by *base*. The size of each member is specified by *size*. The array is sorted in ascending order, according to the results of the function pointed to by *cmp*. *Not implemented because of recursion.*

Integer arithmetic

`int abs(j)`
`long labs(j)`
`long long llabs(j)`
	Compute the absolute value of an `int`, `long int`, and `long long int` *j* respectively.

`div_t div(x,y)`
`ldiv_t ldiv(x,y)`
`lldiv_t lldiv(x,y)`
	Compute *x*/*y* and *x*%*y* in a single operation. *X* and *y* have respectively type `int`, `long int` and `long long int`. The result is stored in the members `quot` and `rem` of `struct div_t`, `ldiv_t` and `lldiv_t` which have the same types.

Multibyte/wide character and string conversions

`mblen(*s,n)`	Determines the number of bytes in the multi-byte character pointed to by *s*. At most *n* characters will be examined. (See also `mbrlen` in Section 5.2.25, *wchar.h*).
`mbtowc(*pwc,*s,n)`	Converts the multi-byte character in *s* to a wide-character code and stores it in pwc. At most *n* characters will be examined.
`wctomb(*s,wc)`	Converts the wide-character *wc* into a multi-byte representation and stores it in the string pointed to by *s*. At most MB_CUR_MAX characters are stored.
`mbstowcs(*pwcs,*s,n)`	Converts a sequence of multi-byte characters in the string pointed to by *s* into a sequence of wide characters and stores at most *n* wide characters into the array pointed to by *pwcs*. (See also `mbsrtowcs` in Section 5.2.25, *wchar.h*).

`wcstombs(*s,*pwcs,n)` Converts a sequence of wide characters in the array pointed to by *pwcs* into multi-byte characters and stores at most *n* multi-byte characters into the string pointed to by *s*. (See also `wcsrtowmb` in Section 5.2.25, *wchar.h*).

5.2.23. string.h and wchar.h

This header file provides numerous functions for manipulating strings. By convention, strings in C are arrays of characters with a terminating null character. Most functions therefore take arguments of type `*char`. However, many functions have also parallel wide-character functions which take arguments of type `*wchar_t`. These functions are declared in `wchar.h`.

Copying and concatenation functions

string.h	wchar.h	Description
`memcpy(*s1,*s2,n)`	`wmemcpy(*s1,*s2,n)`	Copies *n* characters from **s2* into **s1* and returns *s1. If *s1 and *s2 overlap the result is undefined.
`memmove(*s1,*s2,n)`	`wmemmove(*s1,*s2,n)`	Same as `memcpy`, but overlapping strings are handled correctly. Returns **s1*.
`strcpy(*s1,*s2)`	`wcscpy(*s1,*s2)`	Copies **s2* into **s1* and returns **s1*. If **s1* and **s2* overlap the result is undefined.
`strncpy(*s1,*s2,n)`	`wcsncpy(*s1,*s2,n)`	Copies not more than *n* characters from *s2 into **s1* and returns **s1*. If **s1* and **s2* overlap the result is undefined.
`strcat(*s1,*s2)`	`wcscat(*s1,*s2)`	Appends a copy of *s2 to **s1* and returns **s1*. If **s1* and **s2* overlap the result is undefined.
`strncat(*s1,*s2,n)`	`wcsncat(*s1,*s2,n)`	Appends not more than *n* characters from **s2* to **s1* and returns **s1*. If **s1* and **s2* overlap the result is undefined.

Comparison functions

string.h	wchar.h	Description
`memcmp(*s1,*s2,n)`	`wmemcmp(*s1,*s2,n)`	Compares the first *n* characters of **s1* to the first *n* characters of **s2*. Returns < 0 if **s1* < **s2*, 0 if **s1* = = **s2*, or > 0 if **s1* > **s2*.
`strcmp(*s1,*s2)`	`wcscmp(*s1,*s2)`	Compares string **s1* to **s2*. Returns < 0 if **s1* < **s2*, 0 if **s1* = = **s2*, or > 0 if **s1* > **s2*.
`strncmp(*s1,*s2,n)`	`wcsncmp(*s1,*s2,n)`	Compares the first *n* characters of **s1* to the first *n* characters of **s2*. Returns < 0 if **s1* < **s2*, 0 if **s1* = = **s2*, or > 0 if **s1* > **s2*.
`strcoll(*s1,*s2)`	`wcscoll(*s1,*s2)`	Performs a local-specific comparison between string **s1* and string **s2* according to the LC_COLLATE category of the current locale. Returns < 0 if **s1* < **s2*, 0 if **s1* = = **s2*, or > 0 if **s1* > **s2*. (See Section 5.2.12, *locale.h*)

string.h	wchar.h	Description
`strxfrm(*s1,*s2,n)`	`wcsxfrm(*s1,*s2,n)`	Transforms (a local) string *s2 so that a comparison between transformed strings with `strcmp` gives the same result as a comparison between non-transformed strings with `strcoll`. Returns the transformed string *s1.

Search functions

string.h	wchar.h	Description
`memchr(*s,c,n)`	`wmemchr(*s,c,n)`	Checks the first *n* characters of *s on the occurrence of character *c*. Returns a pointer to the found character.
`strchr(*s,c)`	`wcschr(*s,c)`	Returns a pointer to the first occurrence of character *c* in *s or the null pointer if not found.
`strrchr(*s,c)`	`wcsrchr(*s,c)`	Returns a pointer to the last occurrence of character *c* in *s or the null pointer if not found.
`strspn(*s,*set)`	`wcsspn(*s,*set)`	Searches *s for a sequence of characters specified in *set. Returns the length of the first sequence found.
`strcspn(*s,*set)`	`wcscspn(*s,*set)`	Searches *s for a sequence of characters *not* specified in *set. Returns the length of the first sequence found.
`strpbrk(*s,*set)`	`wcspbrk(*s,*set)`	Same as `strspn`/`wcsspn` but returns a pointer to the first character in *s that also is specified in *set.
`strstr(*s,*sub)`	`wcsstr(*s,*sub)`	Searches for a substring *sub in *s. Returns a pointer to the first occurrence of *sub in *s.
`strtok(*s,*dlm)`	`wcstok(*s,*dlm)`	A sequence of calls to this function breaks the string *s into a sequence of tokens delimited by a character specified in *dlm. The token found in *s is terminated with a null character. Returns a pointer to the first position in *s of the token.

Miscellaneous functions

string.h	wchar.h	Description
`memset(*s,c,n)`	`wmemset(*s,c,n)`	Fills the first *n* bytes of *s with character *c* and returns *s.
`strerror(errno)`	-	Typically, the values for errno come from `int errno`. This function returns a pointer to the associated error message. (See also Section 5.2.4, *errno.h*)
`strlen(*s)`	`wcslen(*s)`	Returns the length of string *s.

5.2.24. time.h and wchar.h

The header file `time.h` provides facilities to retrieve and use the (calendar) date and time, and the process time. Time can be represented as an integer value, or can be broken-down in components. Two arithmetic data types are defined which are capable of holding the integer representation of times:

```
clock_t unsigned long long
time_t unsigned long
```

The type `struct tm` below is defined according to ISO C99 with one exception: this implementation does not support leap seconds. The `struct tm` type is defines as follows:

```
struct tm
{
  int   tm_sec;       /* seconds after the minute - [0, 59]   */
  int   tm_min;       /* minutes after the hour - [0, 59]     */
  int   tm_hour;      /* hours since midnight - [0, 23]       */
  int   tm_mday;      /* day of the month - [1, 31]           */
  int   tm_mon;       /* months since January - [0, 11]       */
  int   tm_year;      /* year since 1900                      */
  int   tm_wday;      /* days since Sunday - [0, 6]           */
  int   tm_yday;      /* days since January 1 - [0, 365]      */
  int   tm_isdst;     /* Daylight Saving Time flag            */
};
```

Time manipulation

`clock`	Returns the application's best approximation to the processor time used by the program since it was started. This low-level routine is not implemented because it strongly depends on the hardware. To determine the time in seconds, the result of clock should be divided by the value defined by `CLOCKS_PER_SEC`.
`difftime(`*`t1`*`,`*`t0`*`)`	Returns the difference *t1-t0* in seconds.
`mktime(tm *`*`tp`*`)`	Converts the broken-down time in the structure pointed to by *tp*, to a value of type `time_t`. The return value has the same encoding as the return value of the `time` function.
`time(*`*`timer`*`)`	Returns the current calendar time. This value is also assigned to `*timer`.

Time conversion

`asctime(tm *`*`tp`*`)`	Converts the broken-down time in the structure pointed to by *tp* into a string in the form `Mon Jan 21 16:15:14 2004\n\0`. Returns a pointer to this string.
`ctime(*`*`timer`*`)`	Converts the calender time pointed to by *timer* to local time in the form of a string. This is equivalent to: `asctime(localtime(timer))`
`gmtime(*`*`timer`*`)`	Converts the calender time pointed to by *timer* to the broken-down time, expressed as UTC. Returns a pointer to the broken-down time.
`localtime(*`*`timer`*`)`	Converts the calendar time pointed to by *timer* to the broken-down time, expressed as local time. Returns a pointer to the broken-down time.

Formatted time

The next function has a parallel function defined in `wchar.h`:

time.h	wchar.h
`strftime(*s,smax,*fmt,tm *tp)`	`wstrftime(*s,smax,*fmt,tm *tp)`

Formats date and time information from `struct tm` **tp* into **s* according to the specified format **fmt*. No more than *smax* characters are placed into **s*. The formatting of `strftime` is locale-specific using the `LC_TIME` category (see Section 5.2.12, *locale.h*).

You can use the next conversion specifiers:

%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	locale-specific date and time representation (same as `%a %b %e %T %Y`)
%C	last two digits of the year
%d	day of the month (01-31)
%D	same as `%m/%d/%y`
%e	day of the month (1-31), with single digits preceded by a space
%F	ISO 8601 date format: `%Y-%m-%d`
%g	last two digits of the week based year (00-99)
%G	week based year (0000–9999)
%h	same as %b
%H	hour, 24-hour clock (00-23)
%I	hour, 12-hour clock (01-12)
%j	day of the year (001-366)
%m	month (01-12)
%M	minute (00-59)
%n	replaced by newline character
%p	locale's equivalent of AM or PM
%r	locale's 12-hour clock time; same as `%I:%M:%S %p`
%R	same as `%H:%M`
%S	second (00-59)
%t	replaced by horizontal tab character
%T	ISO 8601 time format: `%H:%M:%S`
%u	ISO 8601 weekday number (1-7), Monday as first day of the week
%U	week number of the year (00-53), week 1 has the first Sunday
%V	ISO 8601 week number (01-53) in the week-based year
%w	weekday (0-6, Sunday is 0)

%W	week number of the year (00-53), week 1 has the first Monday
%x	local date representation
%X	local time representation
%y	year without century (00-99)
%Y	year with century
%z	ISO 8601 offset of time zone from UTC, or nothing
%Z	time zone name, if any
%%	%

5.2.25. wchar.h

Many functions in `wchar.h` represent the wide-character variant of other functions so these are discussed together. (See Section 5.2.21, *stdio.h and wchar.h*, Section 5.2.22, *stdlib.h and wchar.h*, Section 5.2.23, *string.h and wchar.h* and Section 5.2.24, *time.h and wchar.h*).

The remaining functions are described below. They perform conversions between multi-byte characters and wide characters. In these functions, *ps* points to struct `mbstate_t` which holds the conversion state information necessary to convert between sequences of multibyte characters and wide characters:

```
typedef struct
{
     wchar_t         wc_value;  /* wide character value solved
                                   so far */
     unsigned short  n_bytes;   /* number of bytes of solved
                                   multibyte */
     unsigned short  encoding;  /* encoding rule for wide
                                   character <=> multibyte
                                   conversion */
} mbstate_t;
```

When multibyte characters larger than 1 byte are used, this struct will be used to store the conversion information when not all the bytes of a particular multibyte character have been read from the source. In this implementation, multi-byte characters are 1 byte long (MB_CUR_MAX and MB_LEN_MAX are defined as 1) and this will never occur.

`mbsinit(*ps)`	Determines whether the object pointed to by *ps*, is an initial conversion state. Returns a non-zero value if so.
`mbsrtowcs(*pwcs,**src,n,*ps)`	Restartable version of `mbstowcs`. See Section 5.2.22, *stdlib.h and wchar.h*. The initial conversion state is specified by *ps*. The input sequence of multibyte characters is specified indirectly by *src*.
`wcsrtombs(*s,**src,n,*ps)`	Restartable version of `wcstombs`. See Section 5.2.22, *stdlib.h and wchar.h*. The initial conversion state is specified by *ps*. The input wide string is specified indirectly by *src*.
`mbrtowc(*pwc,*s,n,*ps)`	Converts a multibyte character *s to a wide character *pwc according to conversion state *ps*. See also `mbtowc` in Section 5.2.22, *stdlib.h and wchar.h*.

`wcrtomb(*s,wc,*ps)`	Converts a wide character wc to a multi-byte character according to conversion state *ps* and stores the multi-byte character in **s*.
`btowc(c)`	Returns the wide character corresponding to character *c*. Returns WEOF on error.
`wctob(c)`	Returns the multi-byte character corresponding to the wide character *c*. The returned multi-byte character is represented as one byte. Returns EOF on error.
`mbrlen(*s,n,*ps)`	Inspects up to *n* bytes from the string **s* to see if those characters represent valid multibyte characters, relative to the conversion state held in **ps*.

5.2.26. wctype.h

Most functions in `wctype.h` represent the wide-character variant of functions declared in `ctype.h` and are discussed in Section 5.2.3, *ctype.h and wctype.h*. In addition, this header file provides extensible, locale specific functions and wide character classification.

`wctype(*property)`	Constructs a value of type `wctype_t` that describes a class of wide characters identified by the string **property*. If property identifies a valid class of wide characters according to the LC_TYPE category (see Section 5.2.12, *locale.h*) of the current locale, a non-zero value is returned that can be used as an argument in the `iswctype` function.
`iswctype(wc,desc)`	Tests whether the wide character *wc* is a member of the class represented by `wctype_t` *desc*. Returns a non-zero value if tested true.

Function	Equivalent to locale specific test
`iswalnum(wc)`	`iswctype(wc,wctype("alnum"))`
`iswalpha(wc)`	`iswctype(wc,wctype("alpha"))`
`iswcntrl(wc)`	`iswctype(wc,wctype("cntrl"))`
`iswdigit(wc)`	`iswctype(wc,wctype("digit"))`
`iswgraph(wc)`	`iswctype(wc,wctype("graph"))`
`iswlower(wc)`	`iswctype(wc,wctype("lower"))`
`iswprint(wc)`	`iswctype(wc,wctype("print"))`
`iswpunct(wc)`	`iswctype(wc,wctype("punct"))`
`iswspace(wc)`	`iswctype(wc,wctype("space"))`
`iswupper(wc)`	`iswctype(wc,wctype("upper"))`
`iswxditig(wc)`	`iswctype(wc,wctype("xdigit"))`

`wctrans(*property)`	Constructs a value of type `wctype_t` that describes a mapping between wide characters identified by the string **property*. If property identifies a valid mapping of wide characters according to the LC_TYPE category (see Section 5.2.12, *locale.h*) of the current locale, a non-zero value is returned that can be used as an argument in the `towctrans` function.

`towctrans(wc,desc)` Transforms wide character *wc* into another wide-character, described by *desc*.

Function	Equivalent to locale specific transformation
`towlower(wc)`	`towctrans(wc,wctrans("tolower")`
`towupper(wc)`	`towctrans(wc,wctrans("toupper")`

5.3. C Library Reentrancy

The current version of the CHC compiler does not support reentrancy.

Some of the functions in the C library are reentrant, others are not. The table below shows the functions in the C library, and whether they are reentrant or not. A dash means that the function is reentrant. Note that some of the functions are not reentrant because they set the global variable 'errno' (or call other functions that eventually set 'errno'). If your program does not check this variable and errno is the only reason for the function not being reentrant, these functions can be assumed reentrant as well.

The explanation of the cause why a function is not reentrant sometimes refers to a footnote because the explanation is to lengthy for the table.

Function	Not reentrant because
`_close`	Uses global File System Simulation buffer, fss_buffer
`_doflt`	Uses I/O functions which modify iob[ ]. See (1).
`_doprint`	Uses indirect access to static iob[ ] array. See (1).
`_doscan`	Uses indirect access to iob[ ] and calls ungetc (access to local static ungetc[ ] buffer). See (1).
`_Exit`	See exit.
`_filbuf`	Uses iob[ ]. See (1).
`_flsbuf`	Uses iob[ ]. See (1).
`_getflt`	Uses iob[ ]. See (1).
`_iob`	Defines static iob[ ]. See (1).
`_lseek`	Uses global File System Simulation buffer, _fss_buffer
`_open`	Uses global File System Simulation buffer, _fss_buffer
`_read`	Uses global File System Simulation buffer, _fss_buffer
`_unlink`	Uses global File System Simulation buffer, _fss_buffer
`_write`	Uses global File System Simulation buffer, _fss_buffer
`abort`	Calls exit
`abs labs llabs`	-
`access`	Uses global File System Simulation buffer, _fss_buffer
`acos acosf acosl`	Sets errno.
`acosh acoshf acoshl`	Sets errno via calls to other functions.
`asctime`	asctime defines static array for broken-down time string.

Function	Not reentrant because
asin asinf asinl	Sets errno.
asinh asinhf asinhl	Sets errno via calls to other functions.
atan atanf atanl	-
atan2 atan2f atan2l	-
atanh atanhf atanhl	Sets errno via calls to other functions.
atexit	atexit defines static array with function pointers to execute at exit of program.
atof	-
atoi	-
atol	-
bsearch	-
btowc	-
cabs cabsf cabsl	Sets errno via calls to other functions.
cacos cacosf cacosl	Sets errno via calls to other functions.
cacosh cacosh cfacoshl	Sets errno via calls to other functions.
calloc	calloc uses static buffer management structures. See malloc (5).
carg cargf cargl	-
casin casinf casinl	Sets errno via calls to other functions.
casinh casinh cfasinhl	Sets errno via calls to other functions.
catan catanf catanl	Sets errno via calls to other functions.
catanh catanhf catanhl	Sets errno via calls to other functions.
cbrt cbrtf cbrtl	(*Not implemented*)
ccos ccosf ccosl	Sets errno via calls to other functions.
ccosh ccoshf ccoshl	Sets errno via calls to other functions.
ceil ceilf ceill	-
cexp cexpf cexpl	Sets errno via calls to other functions.
chdir	Uses global File System Simulation buffer, fss_buffer
cimag cimagf cimagl	-
cleanup	Calls fclose. See (1)
clearerr	Modifies iob[ ]. See (1)
clock	-
clog clogf clogl	Sets errno via calls to other functions.
close	Calls _close
conj conjf conjl	-
copysign copysignf copysignl	-

Function	Not reentrant because
`cos cosf cosl`	-
`cosh coshf coshl`	cosh calls exp(), which sets errno. If errno is discarded, cosh is reentrant.
`cpow cpowf cpowl`	Sets errno via calls to other functions.
`cproj cprojf cprojl`	-
`creal crealf creall`	-
`csin csinf csinl`	Sets errno via calls to other functions.
`csinh csinhf csinhl`	Sets errno via calls to other functions.
`csqrt csqrtf csqrtl`	Sets errno via calls to other functions.
`ctan ctanf ctanl`	Sets errno via calls to other functions.
`ctanh ctanhf ctanhl`	Sets errno via calls to other functions.
`ctime`	Calls asctime
`difftime`	-
`div ldiv lldiv`	-
`erf erfl erff`	(*Not implemented*)
`erfc erfcf erfcl`	(*Not implemented*)
`exit`	Calls fclose indirectly which uses iob[ ] calls functions in _atexit array. See (1). To make exit reentrant kernel support is required.
`exp expf expl`	Sets errno.
`exp2 exp2f exp2l`	(*Not implemented*)
`expm1 expm1f expm1l`	(*Not implemented*)
`fabs fabsf fabsl`	-
`fclose`	Uses values in iob[ ]. See (1).
`fdim fdimf fdiml`	(*Not implemented*)
`feclearexcept`	(*Not implemented*)
`fegetenv`	(*Not implemented*)
`fegetexceptflag`	(*Not implemented*)
`fegetround`	(*Not implemented*)
`feholdexept`	(*Not implemented*)
`feof`	Uses values in iob[ ]. See (1).
`feraiseexcept`	(*Not implemented*)
`ferror`	Uses values in iob[ ]. See (1).
`fesetenv`	(*Not implemented*)
`fesetexceptflag`	(*Not implemented*)
`fesetround`	(*Not implemented*)
`fetestexcept`	(*Not implemented*)

Function	Not reentrant because
`feupdateenv`	(*Not implemented*)
`fflush`	Modifies iob[ ]. See (1).
`fgetc fgetwc`	Uses pointer to iob[ ]. See (1).
`fgetpos`	Sets the variable errno and uses pointer to iob[ ]. See (1) / (2).
`fgets fgetws`	Uses iob[ ]. See (1).
`floor floorf floorl`	-
`fma fmaf fmal`	(*Not implemented*)
`fmax fmaxf fmaxl`	(*Not implemented*)
`fmin fminf fminl`	(*Not implemented*)
`fmod fmodf fmodl`	-
`fopen`	Uses iob[ ] and calls malloc when file open for buffered IO. See (1)
`fpclassify`	-
`fprintf fwprintf`	Uses iob[ ]. See (1).
`fputc fputwc`	Uses iob[ ]. See (1).
`fputs fputws`	Uses iob[ ]. See (1).
`fread`	Calls fgetc. See (1).
`free`	free uses static buffer management structures. See malloc (5).
`freopen`	Modifies iob[ ]. See (1).
`frexp frexpf frexpl`	-
`fscanf fwscanf`	Uses iob[ ]. See (1)
`fseek`	Uses iob[ ] and calls _lseek. Accesses ungetc[ ] array. See (1).
`fsetpos`	Uses iob[ ] and sets errno. See (1) / (2).
`fstat`	(*Not implemented*)
`ftell`	Uses iob[ ] and sets errno. Calls _lseek. See (1) / (2).
`fwrite`	Uses iob[ ]. See (1).
`getc getwc`	Uses iob[ ]. See (1).
`getchar getwchar`	Uses iob[ ]. See (1).
`getcwd`	Uses global File System Simulation buffer, _fss_buffer
`getenv`	Skeleton only.
`gets getws`	Uses iob[ ]. See (1).
`gmtime`	gmtime defines static structure
`hypot hypotf hypotl`	Sets errno via calls to other functions.
`ilogb ilogbf ilogbl`	(*Not implemented*)
`imaxabs`	-
`imaxdiv`	-

Function	Not reentrant because
`isalnum iswalnum`	-
`isalpha iswalpha`	-
`isascii iswascii`	-
`iscntrl iswcntrl`	-
`isdigit iswdigit`	-
`isfinite`	-
`isgraph iswgraph`	-
`isgreater`	-
`isgreaterequal`	-
`isinf`	-
`isless`	-
`islessequal`	-
`islessgreater`	-
`islower iswlower`	-
`isnan`	-
`isnormal`	-
`isprint iswprint`	-
`ispunct iswpunct`	-
`isspace iswspace`	-
`isunordered`	-
`isupper iswupper`	-
`iswalnum`	-
`iswalpha`	-
`iswcntrl`	-
`iswctype`	-
`iswdigit`	-
`iswgraph`	-
`iswlower`	-
`iswprint`	-
`iswpunct`	-
`iswspace`	-
`iswupper`	-
`iswxditig`	-
`isxdigit iswxdigit`	-
`ldexp ldexpf ldexpl`	Sets errno. See (2).

Function	Not reentrant because
`lgamma lgammaf lgammal`	(*Not implemented*)
`llrint lrintf lrintl`	(*Not implemented*)
`llround llroundf llroundl`	(*Not implemented*)
`localeconv`	N.A.; skeleton function
`localtime`	-
`log logf logl`	Sets errno. See (2).
`log10 log10f log10l`	Sets errno via calls to other functions.
`log1p log1pf log1pl`	(*Not implemented*)
`log2 log2f log2l`	(*Not implemented*)
`logb logbf logbl`	(*Not implemented*)
`longjmp`	-
`lrint lrintf lrintl`	(*Not implemented*)
`lround lroundf lroundl`	(*Not implemented*)
`lseek`	Calls _lseek
`lstat`	(*Not implemented*)
`malloc`	Needs kernel support. See (5).
`mblen`	N.A., skeleton function
`mbrlen`	Sets errno.
`mbrtowc`	Sets errno.
`mbsinit`	-
`mbsrtowcs`	Sets errno.
`mbstowcs`	N.A., skeleton function
`mbtowc`	N.A., skeleton function
`memchr wmemchr`	-
`memcmp wmemcmp`	-
`memcpy wmemcpy`	-
`memmove wmemmove`	-
`memset wmemset`	-
`mktime`	-
`modf modff modfl`	-
`nan nanf nanl`	(*Not implemented*)
`nearbyint nearbyintf nearbyintl`	(*Not implemented*)
`nextafter nextafterf nextafterl`	(*Not implemented*)

Function	Not reentrant because
`nexttoward nexttowardf nexttowardl`	(*Not implemented*)
`offsetof`	-
`open`	Calls _open
`perror`	Uses errno. See (2)
`pow powf powl`	Sets errno. See (2)
`printf wprintf`	Uses iob[ ]. See (1)
`putc putwc`	Uses iob[ ]. See (1)
`putchar putwchar`	Uses iob[ ]. See (1)
`puts`	Uses iob[ ]. See (1)
`qsort`	-
`raise`	Updates the signal handler table
`rand`	Uses static variable to remember latest random number. Must diverge from ANSI standard to define reentrant rand. See (4).
`read`	Calls _read
`realloc`	See malloc (5).
`remainder remainderf remainderl`	(*Not implemented*)
`remove`	N.A; skeleton only.
`remquo remquof remquol`	(*Not implemented*)
`rename`	N.A; skeleton only.
`rewind`	Eventually calls _lseek
`rint rintf rintl`	(*Not implemented*)
`round roundf roundl`	(*Not implemented*)
`scalbln scalblnf scalblnl`	-
`scalbn scalbnf scalbnl`	-
`scanf wscanf`	Uses iob[ ], calls _doscan. See (1).
`setbuf`	Sets iob[ ]. See (1).
`setjmp`	-
`setlocale`	N.A.; skeleton function
`setvbuf`	Sets iob and calls malloc. See (1) / (5).
`signal`	Updates the signal handler table
`signbit`	-
`sin sinf sinl`	-
`sinh sinhf sinhl`	Sets errno via calls to other functions.
`snprintf swprintf`	Sets errno. See (2).

Function	Not reentrant because
sprintf	Sets errno. See (2).
sqrt sqrtf sqrtl	Sets errno. See (2).
srand	See rand
sscanf swscanf	Sets errno via calls to other functions.
stat	Uses global File System Simulation buffer, _fss_buffer
strcat wcscat	-
strchr wcschr	-
strcmp wcscmp	-
strcoll wcscoll	-
strcpy wcscpy	-
strcspn wcscspn	-
strerror	-
strftime wstrftime	-
strlen wcslen	-
strncat wcsncat	-
strncmp wcsncmp	-
strncpy wcsncpy	-
strpbrk wcspbrk	-
strrchr wcsrchr	-
strspn wcsspn	-
strstr wcsstr	-
strtod wcstod	-
strtof wcstof	-
strtoimax	Sets errno via calls to other functions.
strtok wcstok	Strtok saves last position in string in local static variable. This function is not reentrant by design. See (4).
strtol wcstol	Sets errno. See (2).
strtold wcstold	-
strtoul wcstoul	Sets errno. See (2).
strtoull wcstoull	Sets errno. See (2).
strtoumax	Sets errno via calls to other functions.
strxfrm wcsxfrm	-
system	N.A; skeleton function
tan tanf tanl	Sets errno. See (2).
tanh tanhf tanhl	Sets errno via call to other functions.

Function	Not reentrant because
`tgamma tgammaf tgammal`	(*Not implemented*)
`time`	Uses static variable which defines initial start time
`tmpfile`	Uses iob[ ]. See (1).
`tmpnam`	Uses local buffer to build filename. Function can be adapted to use user buffer. This makes the function non ANSI. See (4).
`toascii`	-
`tolower`	-
`toupper`	-
`towctrans`	-
`towlower`	-
`towupper`	-
`trunc truncf truncl`	(*Not implemented*)
`ungetc ungetwc`	Uses static buffer to hold unget characters for each file. Can be moved into iob structure. See (1).
`unlink`	Calls _unlink
`vfprintf vfwprintf`	Uses iob[ ]. See (1).
`vfscanf vfwscanf`	Calls _doscan
`vprintf vwprintf`	Uses iob[ ]. See (1).
`vscanf vwscanf`	Calls _doscan
`vsprintf vswprintf`	Sets errno.
`vsscanf vswscanf`	Sets errno.
`wcrtomb`	Sets errno.
`wcsrtombs`	Sets errno.
`wcstoimax`	Sets errno via calls to other functions.
`wcstombs`	N.A.; skeleton function
`wcstoumax`	Sets errno via calls to other functions.
`wctob`	-
`wctomb`	N.A.; skeleton function
`wctrans`	-
`wctype`	-
`write`	Calls _write

Table: C library reentrancy

Several functions in the C library are not reentrant due to the following reasons:

- The `iob[ ]` structure is static. This influences all I/O functions.

- The `ungetc[ ]` array is static. This array holds the characters (one for each stream) when `ungetc( )` is called.
- The variable `errno` is globally defined. Numerous functions read or modify `errno`
- `_doprint` and `_doscan` use static variables for e.g. character counting in strings.
- Some string functions use locally defined (static) buffers. This is prescribed by ANSI.
- `malloc` uses a static heap space.

The following description discusses these items into more detail. The numbers at the begin of each paragraph relate to the number references in the table above.

(1) iob structures

The I/O part of the C library is not reentrant by design. This is mainly caused by the static declaration of the iob[ ] array. The functions which use elements of this array access these elements via pointers (`FILE *`).

Building a multi-process system that is created in one link-run is hard to do. The C language scoping rules for external variables make it difficult to create a private copy of the `iob[ ]` array. Currently, the `iob[ ]` array has external scope. Thus it is visible in every module involved in one link phase. If these modules comprise several tasks (processes) in a system each of which should have its private copy of `iob[ ]`, it is apparent that the `iob[ ]` declaration should be changed. This requires adaptation of the library to the multi-tasking environment. The library modules must use a process identification as an index for determining which `iob[ ]` array to use. Thus the library is suitable for interfacing to that kernel only.

Another approach for the `iob[ ]` declaration problem is to declare the array static in one of the modules which create a task. Thus there can be more than one `iob[ ]` array is the system without having conflicts at link time. This brings several restrictions: Only the module that holds the declaration of the static `iob[ ]` can use the standard file handles `stdin`, `stdout` and `stderr` (which are the first three entries in `iob[ ]`). Thus all I/O for these three file handles should be located in one module.

(2) errno declaration

Several functions in the C library set the global variable `errno`. After completion of the function the user program may consult this variable to see if some error occurred. Since most of the functions that set `errno` already have a return type (this is the reason for using `errno`) it is not possible to check successful completion via the return type.

The library routines can set `errno` to the values defined in `errno.h`. See the file `errno.h` for more information.

`errno` can be set to ERR_FORMAT by the print and scan functions in the C library if you specify illegal format strings.

`errno` will never be set to ERR_NOLONG or ERR_NOPOINT since the CHC C library supports long and pointer conversion routines for input and output.

`errno` can be set to ERANGE by the following functions: `exp()`, `strtol()`, `strtoul()` and `tan()`. These functions may produce results that are out of the valid range for the return type. If so, the result of the function will be the largest representable value for that type and `errno` is set to ERANGE.

`errno` is set to EDOM by the following functions: `acos()`, `asin()`, `log()`, `pow()` and `sqrt()`. If the arguments for these functions are out of their valid range (e.g. `sqrt( -1 )`), `errno` is set to EDOM.

`errno` can be set to ERR_POS by the file positioning functions `ftell()`, `fsetpos()` and `fgetpos()`.

(3) ungetc

Currently the ungetc buffer is static. For each file entry in the `iob[ ]` structure array, there is one character available in the buffer to unget a character.

(4) local buffers

`tmpnam()` creates a temporary filename and returns a pointer to a local static buffer. This is according to the ANSI definition. Changing this function such that it creates the name in a user specified buffer requires another calling interface. Thus the function would be no longer portable.

`strtok()` scans through a string and remembers that the string and the position in the string for subsequent calls. This function is not reentrant by design. Making it reentrant requires support of a kernel to store the information on a per process basis.

`rand()` generates a sequence of random numbers. The function uses the value returned by a previous call to generate the next value in the sequence. This function can be made reentrant by specifying the previous random value as one of the arguments. However, then it is no longer a standard function.

(5) malloc

Malloc uses a heap space which is assigned at locate time. Thus this implementation is not reentrant. Making a reentrant malloc requires some sort of system call to obtain free memory space on a per process basis. This is not easy to solve within the current context of the library. This requires adaptation to a kernel.

This paragraph on reentrancy applies to multi-process environments only. If reentrancy is required for calling library functions from an exception handler, another approach is required. For such a situation it is of no use to allocate e.g. multiple `iob[ ]` structures. In such a situation several pieces of code in the library have to be declared 'atomic': this means that interrupts have to be disabled while executing an atomic piece of code.

Chapter 6. MISRA-C Rules

This chapter contains an overview of the supported and unsupported MISRA C rules.

6.1. MISRA-C:1998

This section lists all supported and unsupported MISRA-C:1998 rules.

See also Section 4.8, *C Code Checking: MISRA-C*.

A number of MISRA-C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

x means that the rule is not supported by the C compiler. (R) is a required rule, (A) is an advisory rule.

	1.	(R)	The code shall conform to standard C, without language extensions
x	2.	(A)	Other languages should only be used with an interface standard
	3.	(A)	Inline assembly is only allowed in dedicated C functions
x	4.	(A)	Provision should be made for appropriate run-time checking
	5.	(R)	Only use characters and escape sequences defined by ISO C
x	6.	(R)	Character values shall be restricted to a subset of ISO 106460-1
	7.	(R)	Trigraphs shall not be used
	8.	(R)	Multibyte characters and wide string literals shall not be used
	9.	(R)	Comments shall not be nested
	10.	(A)	Sections of code should not be "commented out" In general, it is not possible to decide whether a piece of comment is C code that is commented out, or just some pseudo code. Instead, the following heuristics are used to detect possible C code inside a comment: • a line ends with ';', or • a line starts with '}', possibly preceded by white space
	11.	(R)	Identifiers shall not rely on significance of more than 31 characters
	12.	(A)	The same identifier shall not be used in multiple name spaces
	13.	(A)	Specific-length typedefs should be used instead of the basic types
	14.	(R)	Use 'unsigned char' or 'signed char' instead of plain 'char'
x	15.	(A)	Floating-point implementations should comply with a standard
	16.	(R)	The bit representation of floating-point numbers shall not be used A violation is reported when a pointer to a floating-point type is converted to a pointer to an integer type.
	17.	(R)	"typedef" names shall not be reused

	18.	(A)	Numeric constants should be suffixed to indicate type A violation is reported when the value of the constant is outside the range indicated by the suffixes, if any.
	19.	(R)	Octal constants (other than zero) shall not be used
	20.	(R)	All object and function identifiers shall be declared before use
	21.	(R)	Identifiers shall not hide identifiers in an outer scope
	22.	(A)	Declarations should be at function scope where possible
x	23.	(A)	All declarations at file scope should be static where possible
	24.	(R)	Identifiers shall not have both internal and external linkage
x	25.	(R)	Identifiers with external linkage shall have exactly one definition
	26.	(R)	Multiple declarations for objects or functions shall be compatible
x	27.	(A)	External objects should not be declared in more than one file
	28.	(A)	The "register" storage class specifier should not be used
	29.	(R)	The use of a tag shall agree with its declaration
	30.	(R)	All automatics shall be initialized before being used This rule is checked using worst-case assumptions. This means that violations are reported not only for variables that are guaranteed to be uninitialized, but also for variables that are uninitialized on some execution paths.
	31.	(R)	Braces shall be used in the initialization of arrays and structures
	32.	(R)	Only the first, or all enumeration constants may be initialized
	33.	(R)	The right hand operand of && or || shall not contain side effects
	34.	(R)	The operands of a logical && or || shall be primary expressions
	35.	(R)	Assignment operators shall not be used in Boolean expressions
	36.	(A)	Logical operators should not be confused with bitwise operators
	37.	(R)	Bitwise operations shall not be performed on signed integers
	38.	(R)	A shift count shall be between 0 and the operand width minus 1 This violation will only be checked when the shift count evaluates to a constant value at compile time.
	39.	(R)	The unary minus shall not be applied to an unsigned expression
	40.	(A)	"sizeof" should not be used on expressions with side effects
x	41.	(A)	The implementation of integer division should be documented
	42.	(R)	The comma operator shall only be used in a "for" condition
	43.	(R)	Don't use implicit conversions which may result in information loss
	44.	(A)	Redundant explicit casts should not be used
	45.	(R)	Type casting from any type to or from pointers shall not be used
	46.	(R)	The value of an expression shall be evaluation order independent This rule is checked using worst-case assumptions. This means that a violation will be reported when a possible alias may cause the result of an expression to be evaluation order dependent.
	47.	(A)	No dependency should be placed on operator precedence rules

48. (A) Mixed arithmetic should use explicit casting
49. (A) Tests of a (non-Boolean) value against 0 should be made explicit
50. (R) F.P. variables shall not be tested for exact equality or inequality
51. (A) Constant unsigned integer expressions should not wrap-around
52. (R) There shall be no unreachable code
53. (R) All non-null statements shall have a side-effect
54. (R) A null statement shall only occur on a line by itself
55. (A) Labels should not be used
56. (R) The "goto" statement shall not be used
57. (R) The "continue" statement shall not be used
58. (R) The "break" statement shall not be used (except in a "switch")
59. (R) An "if" or loop body shall always be enclosed in braces
60. (A) All "if", "else if" constructs should contain a final "else"
61. (R) Every non-empty "case" clause shall be terminated with a "break"
62. (R) All "switch" statements should contain a final "default" case
63. (A) A "switch" expression should not represent a Boolean case
64. (R) Every "switch" shall have at least one "case"
65. (R) Floating-point variables shall not be used as loop counters
66. (A) A "for" should only contain expressions concerning loop control
    A violation is reported when the loop initialization or loop update expression modifies an object that is not referenced in the loop test.
67. (A) Iterator variables should not be modified in a "for" loop
68. (R) Functions shall always be declared at file scope
69. (R) Functions with variable number of arguments shall not be used
70. (R) Functions shall not call themselves, either directly or indirectly
    A violation will be reported for direct or indirect recursive function calls in the source file being checked. Recursion via functions in other source files, or recursion via function pointers is not detected.
71. (R) Function prototypes shall be visible at the definition and call
72. (R) The function prototype of the declaration shall match the definition
73. (R) Identifiers shall be given for all prototype parameters or for none
74. (R) Parameter identifiers shall be identical for declaration/definition
75. (R) Every function shall have an explicit return type
76. (R) Functions with no parameters shall have a "void" parameter list
77. (R) An actual parameter type shall be compatible with the prototype
78. (R) The number of actual parameters shall match the prototype
79. (R) The values returned by "void" functions shall not be used
80. (R) Void expressions shall not be passed as function parameters

81. (A) "const" should be used for reference parameters not modified
82. (A) A function should have a single point of exit
83. (R) Every exit point shall have a "return" of the declared return type
84. (R) For "void" functions, "return" shall not have an expression
85. (A) Function calls with no parameters should have empty parentheses
86. (A) If a function returns error information, it should be tested
    A violation is reported when the return value of a function is ignored.
87. (R) #include shall only be preceded by other directives or comments
88. (R) Non-standard characters shall not occur in #include directives
89. (R) #include shall be followed by either <filename> or "filename"
90. (R) Plain macros shall only be used for constants/qualifiers/specifiers
91. (R) Macros shall not be #define'd and #undef'd within a block
92. (A) #undef should not be used
93. (A) A function should be used in preference to a function-like macro
94. (R) A function-like macro shall not be used without all arguments
95. (R) Macro arguments shall not contain pre-preprocessing directives
    A violation is reported when the first token of an actual macro argument is '#'.
96. (R) Macro definitions/parameters should be enclosed in parentheses
97. (A) Don't use undefined identifiers in pre-processing directives
98. (R) A macro definition shall contain at most one # or ## operator
99. (R) All uses of the #pragma directive shall be documented
    This rule is really a documentation issue. The compiler will flag all #pragma directives as violations.
100. (R) "defined" shall only be used in one of the two standard forms
101. (A) Pointer arithmetic should not be used
102. (A) No more than 2 levels of pointer indirection should be used
    A violation is reported when a pointer with three or more levels of indirection is declared.
103. (R) No relational operators between pointers to different objects
    In general, checking whether two pointers point to the same object is impossible. The compiler will only report a violation for a relational operation with incompatible pointer types.
104. (R) Non-constant pointers to functions shall not be used
105. (R) Functions assigned to the same pointer shall be of identical type
106. (R) Automatic address may not be assigned to a longer lived object
107. (R) The null pointer shall not be dereferenced
    A violation is reported for every pointer dereference that is not guarded by a NULL pointer test.
108. (R) All struct/union members shall be fully specified

	109.	(R)	Overlapping variable storage shall not be used A violation is reported for every 'union' declaration.
	110.	(R)	Unions shall not be used to access the sub-parts of larger types A violation is reported for a 'union' containing a 'struct' member.
	111.	(R)	bit-fields shall have type "unsigned int" or "signed int"
	112.	(R)	bit-fields of type "signed int" shall be at least 2 bits long
	113.	(R)	All struct/union members shall be named
	114.	(R)	Reserved and standard library names shall not be redefined
	115.	(R)	Standard library function names shall not be reused
x	116.	(R)	Production libraries shall comply with the MISRA C restrictions
x	117.	(R)	The validity of library function parameters shall be checked
	118.	(R)	Dynamic heap memory allocation shall not be used
	119.	(R)	The error indicator "errno" shall not be used
	120.	(R)	The macro "offsetof" shall not be used
	121.	(R)	<locale.h> and the "setlocale" function shall not be used
	122.	(R)	The "setjmp" and "longjmp" functions shall not be used
	123.	(R)	The signal handling facilities of <signal.h> shall not be used
	124.	(R)	The <stdio.h> library shall not be used in production code
	125.	(R)	The functions atof/atoi/atol shall not be used
	126.	(R)	The functions abort/exit/getenv/system shall not be used
	127.	(R)	The time handling functions of library <time.h> shall not be used

6.2. MISRA-C:2004

This section lists all supported and unsupported MISRA-C:2004 rules.

See also Section 4.8, *C Code Checking: MISRA-C*.

A number of MISRA-C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

x means that the rule is not supported by the C compiler. (R) is a required rule, (A) is an advisory rule.

Environment

	1.1	(R)	All code shall conform to ISO 9899:1990 "Programming languages - C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.
	1.2	(R)	No reliance shall be placed on undefined or unspecified behavior.
x	1.3	(R)	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform.

x	1.4	(R)	The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.
x	1.5	(A)	Floating-point implementations should comply with a defined floating-point standard.

Language extensions

	2.1	(R)	Assembly language shall be encapsulated and isolated.
	2.2	(R)	Source code shall only use `/* ... */` style comments.
	2.3	(R)	The character sequence `/*` shall not be used within a comment.
	2.4	(A)	Sections of code should not be "commented out". In general, it is not possible to decide whether a piece of comment is C code that is commented out, or just some pseudo code. Instead, the following heuristics are used to detect possible C code inside a comment: - a line ends with ';', or - a line starts with '}', possibly preceded by white space

Documentation

x	3.1	(R)	All usage of implementation-defined behavior shall be documented.
x	3.2	(R)	The character set and the corresponding encoding shall be documented.
x	3.3	(A)	The implementation of integer division in the chosen compiler should be determined, documented and taken into account.
	3.4	(R)	All uses of the `#pragma` directive shall be documented and explained. This rule is really a documentation issue. The compiler will flag all `#pragma` directives as violations.
	3.5	(R)	The implementation-defined behavior and packing of bit-fields shall be documented if being relied upon.
x	3.6	(R)	All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.

Character sets

	4.1	(R)	Only those escape sequences that are defined in the ISO C standard shall be used.
	4.2	(R)	Trigraphs shall not be used.

Identifiers

	5.1	(R)	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
	5.2	(R)	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
	5.3	(R)	A `typedef` name shall be a unique identifier.
	5.4	(R)	A tag name shall be a unique identifier.
x	5.5	(A)	No object or function identifier with static storage duration should be reused.

	5.6	(A)	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.
x	5.7	(A)	No identifier name should be reused.

Types

	6.1	(R)	The plain `char` type shall be used only for storage and use of character values.
x	6.2	(R)	`signed` and `unsigned char` type shall be used only for the storage and use of numeric values.
	6.3	(A)	`typedefs` that indicate size and signedness should be used in place of the basic types.
	6.4	(R)	bit-fields shall only be defined to be of type `unsigned int` or `signed int`.
	6.5	(R)	bit-fields of type `signed int` shall be at least 2 bits long.

Constants

	7.1	(R)	Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and definitions

	8.1	(R)	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
	8.2	(R)	Whenever an object or function is declared or defined, its type shall be explicitly stated.
	8.3	(R)	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.
	8.4	(R)	If objects or functions are declared more than once their types shall be compatible.
	8.5	(R)	There shall be no definitions of objects or functions in a header file.
	8.6	(R)	Functions shall be declared at file scope.
	8.7	(R)	Objects shall be defined at block scope if they are only accessed from within a single function.
x	8.8	(R)	An external object or function shall be declared in one and only one file.
x	8.9	(R)	An identifier with external linkage shall have exactly one external definition.
x	8.10	(R)	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.
	8.11	(R)	The `static` storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
	8.12	(R)	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

Initialization

9.1 (R) All automatic variables shall have been assigned a value before being used. This rule is checked using worst-case assumptions. This means that violations are reported not only for variables that are guaranteed to be uninitialized, but also for variables that are uninitialized on some execution paths.

9.2 (R) Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

9.3 (R) In an enumerator list, the "=" construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Arithmetic type conversions

10.1 (R) The value of an expression of integer type shall not be implicitly converted to a different underlying type if:
a) it is not a conversion to a wider integer type of the same signedness, or
b) the expression is complex, or
c) the expression is not constant and is a function argument, or
d) the expression is not constant and is a return expression.

10.2 (R) The value of an expression of floating type shall not be implicitly converted to a different type if:
a) it is not a conversion to a wider floating type, or
b) the expression is complex, or
c) the expression is a function argument, or
d) the expression is a return expression.

10.3 (R) The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.

10.4 (R) The value of a complex expression of floating type may only be cast to a narrower floating type.

10.5 (R) If the bitwise operators ~ and << are applied to an operand of underlying type `unsigned char` or `unsigned short`, the result shall be immediately cast to the underlying type of the operand.

10.6 (R) A "U" suffix shall be applied to all constants of `unsigned` type.

Pointer type conversions

11.1 (R) Conversions shall not be performed between a pointer to a function and any type other than an integral type.

11.2 (R) Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.

11.3 (A) A cast should not be performed between a pointer type and an integral type.

11.4 (A) A cast should not be performed between a pointer to object type and a different pointer to object type.

11.5 (R) A cast shall not be performed that removes any `const` or `volatile` qualification from the type addressed by a pointer.

Expressions

12.1 (A) Limited dependency should be placed on C's operator precedence rules in expressions.

12.2 (R) The value of an expression shall be the same under any order of evaluation that the standard permits. This rule is checked using worst-case assumptions. This means that a violation will be reported when a possible alias may cause the result of an expression to be evaluation order dependent.

12.3 (R) The `sizeof` operator shall not be used on expressions that contain side effects.

12.4 (R) The right-hand operand of a logical `&&` or `||` operator shall not contain side effects.

12.5 (R) The operands of a logical `&&` or `||` shall be *primary-expressions*.

12.6 (A) The operands of logical operators (`&&`, `||` and `!`) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (`&&`, `||` and `!`).

12.7 (R) Bitwise operators shall not be applied to operands whose underlying type is signed.

12.8 (R) The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand. This violation will only be checked when the shift count evaluates to a constant value at compile time.

12.9 (R) The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

12.10 (R) The comma operator shall not be used.

12.11 (A) Evaluation of constant unsigned integer expressions should not lead to wrap-around.

12.12 (R) The underlying bit representations of floating-point values shall not be used. A violation is reported when a pointer to a floating-point type is converted to a pointer to an integer type.

12.13 (A) The increment (`++`) and decrement (`--`) operators should not be mixed with other operators in an expression.

Control statement expressions

13.1 (R) Assignment operators shall not be used in expressions that yield a Boolean value.

13.2 (A) Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.

13.3 (R) Floating-point expressions shall not be tested for equality or inequality.

13.4 (R) The controlling expression of a `for` statement shall not contain any objects of floating type.

13.5 (R) The three expressions of a `for` statement shall be concerned only with loop control. A violation is reported when the loop initialization or loop update expression modifies an object that is not referenced in the loop test.

13.6 (R) Numeric variables being used within a `for` loop for iteration counting shall not be modified in the body of the loop.

13.7 (R) Boolean operations whose results are invariant shall not be permitted.

Control flow

14.1 (R) There shall be no unreachable code.

14.2 (R) All non-null statements shall either:
a) have at least one side effect however executed, or
b) cause control flow to change.

14.3 (R) Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a whitespace character.

14.4 (R) The `goto` statement shall not be used.

14.5 (R) The `continue` statement shall not be used.

14.6 (R) For any iteration statement there shall be at most one break statement used for loop termination.

14.7 (R) A function shall have a single point of exit at the end of the function.

14.8 (R) The statement forming the body of a `switch`, `while`, `do ... while` or `for` statement be a compound statement.

14.9 (R) An `if (`*`expression`*`)` construct shall be followed by a compound statement. The `else` keyword shall be followed by either a compound statement, or another `if` statement.

14.10 (R) All `if ... else if` constructs shall be terminated with an `else` clause.

Switch statements

15.1 (R) A switch label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement.

15.2 (R) An unconditional `break` statement shall terminate every non-empty `switch` clause.

15.3 (R) The final clause of a switch statement shall be the `default` clause.

15.4 (R) A `switch` expression shall not represent a value that is effectively Boolean.

15.5 (R) Every `switch` statement shall have at least one `case` clause.

Functions

16.1 (R) Functions shall not be defined with variable numbers of arguments.

16.2 (R) Functions shall not call themselves, either directly or indirectly. A violation will be reported for direct or indirect recursive function calls in the source file being checked. Recursion via functions in other source files, or recursion via function pointers is not detected.

16.3 (R) Identifiers shall be given for all of the parameters in a function prototype declaration.

16.4 (R) The identifiers used in the declaration and definition of a function shall be identical.

16.5 (R) Functions with no parameters shall be declared with parameter type `void`.

16.6 (R) The number of arguments passed to a function shall match the number of parameters.

	16.7	(A)	A pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.
	16.8	(R)	All exit paths from a function with non-void return type shall have an explicit `return` statement with an expression.
	16.9	(R)	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty.
	16.10	(R)	If a function returns error information, then that error information shall be tested. A violation is reported when the return value of a function is ignored.

Pointers and arrays

x	17.1	(R)	Pointer arithmetic shall only be applied to pointers that address an array or array element.
x	17.2	(R)	Pointer subtraction shall only be applied to pointers that address elements of the same array.
	17.3	(R)	>, >=, <, <= shall not be applied to pointer types except where they point to the same array. In general, checking whether two pointers point to the same object is impossible. The compiler will only report a violation for a relational operation with incompatible pointer types.
	17.4	(R)	Array indexing shall be the only allowed form of pointer arithmetic.
	17.5	(A)	The declaration of objects should contain no more than 2 levels of pointer indirection. A violation is reported when a pointer with three or more levels of indirection is declared.
	17.6	(R)	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Structures and unions

	18.1	(R)	All structure or union types shall be complete at the end of a translation unit.
	18.2	(R)	An object shall not be assigned to an overlapping object.
x	18.3	(R)	An area of memory shall not be reused for unrelated purposes.
	18.4	(R)	Unions shall not be used.

Preprocessing directives

	19.1	(A)	`#include` statements in a file should only be preceded by other preprocessor directives or comments.
	19.2	(A)	Non-standard characters should not occur in header file names in `#include` directives.
x	19.3	(R)	The `#include` directive shall be followed by either a <filename> or "filename" sequence.
	19.4	(R)	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.

	19.5	(R)	Macros shall not be `#define`'d or `#undef`'d within a block.
	19.6	(R)	`#undef` shall not be used.
	19.7	(A)	A function should be used in preference to a function-like macro.
	19.8	(R)	A function-like macro shall not be invoked without all of its arguments.
	19.9	(R)	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives. A violation is reported when the first token of an actual macro argument is '#'.
	19.10	(R)	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.
	19.11	(R)	All macro identifiers in preprocessor directives shall be defined before use, except in `#ifdef` and `#ifndef` preprocessor directives and the `defined()` operator.
	19.12	(R)	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.
	19.13	(A)	The # and ## preprocessor operators should not be used.
	19.14	(R)	The `defined` preprocessor operator shall only be used in one of the two standard forms.
	19.15	(R)	Precautions shall be taken in order to prevent the contents of a header file being included twice.
	19.16	(R)	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.
	19.17	(R)	All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` directive to which they are related.

Standard libraries

	20.1	(R)	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.
	20.2	(R)	The names of standard library macros, objects and functions shall not be reused.
x	20.3	(R)	The validity of values passed to library functions shall be checked.
	20.4	(R)	Dynamic heap memory allocation shall not be used.
	20.5	(R)	The error indicator `errno` shall not be used.
	20.6	(R)	The macro `offsetof`, in library `<stddef.h>`, shall not be used.
	20.7	(R)	The `setjmp` macro and the `longjmp` function shall not be used.
	20.8	(R)	The signal handling facilities of `<signal.h>` shall not be used.
	20.9	(R)	The input/output library `<stdio.h>` shall not be used in production code.
	20.10	(R)	The library functions `atof`, `atoi` and `atol` from library `<stdlib.h>` shall not be used.
	20.11	(R)	The library functions `abort`, `exit`, `getenv` and `system` from library `<stdlib.h>` shall not be used.
	20.12	(R)	The time handling functions of library `<time.h>` shall not be used.

Run-time failures

x 21.1 (R) Minimization of run-time failures shall be ensured by the use of at least one of:
a) static analysis tools/techniques;
b) dynamic analysis tools/techniques;
c) explicit coding of checks to handle run-time faults.

Chapter 7. Glossary

ASP	Application Specific Processor. The ASP is placed as a component (WB_ASP) on the FPGA design. This component can contain one or more hardware functions.
CHC Compiler	CHC is an acronym for C-to-Hardware Compiler; a C-to-RTL compiler developed and distributed by Altium. In this manual, *C-to-Hardware Compiler* and *CHC compiler* are used both. The name of this compiler on the command line is **chc**.
C-to-RTL compiler (C-to-Hardware compiler)	The term C-to-RTL compiler is commonly used to identify the class of C compilers that translate C source code into an electronic circuit. The output of the compiler is typically a file that describes the circuit at the RTL level in VHDL or Verilog.
Embedded compiler	The term embedded compiler is used to identify a traditional C compiler that translates a C program into a sequence of instructions that are executed by a microcontroller or DSP.
HASM	Hardware Assembly Language. HASM is a language for describing digital electronic circuits and is the hardware equivalent of normal assembly language. HASM is generated by the **chc** compiler; absolule HASM files are converted to VHDL or Verilog by the **hdlhc** hardware language generator.
HDL	In electronics, a hardware description language or HDL is any language from a class of computer languages for formal description of electronic circuits. It can describe the circuit's operation, its design, and tests to verify its operation by means of simulation.
Hardware function	A C function that is instantiated as an electronic circuit. In the context of an FPGA design, also called an *Application Specific Processor* (ASP). Opposite of software function.
HW-SW mode	An operating mode of the C-to-Hardware Compiler where the C source code is partially translated into an electronic circuit and partially into an instruction sequence that is processed by a processor core. The electronic circuit is build by the hardware compiler whereas the instruction sequence is generated via a traditional embedded compile-assemble-link-locate design flow.
MIL	The *Medium Level Intermediate Language*, is a language used by TASKING compilers to represent the source code in a format that is suited for code generation by the compiler back-end.
RTL	Register transfer level description, also called register transfer logic is a description of a digital electronic circuit in terms of data flow between registers, which store information between clock cycles in a digital circuit. The RTL description specifies what and where this

	information is stored and how it is passed through the circuit during its operation.
Software function	A C function that is executed by a processor core. Opposite of hardware function.
Verilog	Verilog is a hardware description language (HDL) used to model electronic systems. The language (sometimes called Verilog HDL) supports the design, testing, and implementation of analog, digital, and mixed-signal circuits at various levels of abstraction.
VHDL	VHDL or VHSIC Hardware Description Language, is commonly used as a design-entry language for field-programmable gate arrays and application-specific integrated circuits in electronic design automation of digital circuits.
Wishbone Bus	The Wishbone Bus is an open standard hardware computer bus intended to let the parts of an integrated circuit communicate with each other. The aim is to allow the connection of differing cores to each other or to peripheral devices inside of a chip.