6.1 INTRODUCTION

c166 comes with libraries per memory model and with header files containing the appropriate prototype of the library functions. The library functions are also shipped in source code (C or assembly).

Seven sets of libraries are delivered to meet specific requirements for the various C166, Gold, C167/ST10x167/ST10x262, C166S v2.0 and Super10 microcontroller architectures. These sets are located in separate directories:

166	The non–protected libraries are the default libraries for the C166/ST10x166 and similar architectures.
166p	The protected libraries provide a software workaround for CPU functional problems. They must be using in conjunction with the appropriate **–B** compiler option. For more details refer to appendix G: *CPU Functional Problems* for more information.
goldp	These protected libraries are the default libraries for the Gold and similar architectures which are based on C166/ST10x166 architectures but feature 24–bit extended addressing instead of 18–bit. Use these libraries in conjunction with the compiler option **–xm**.
ext	The extended libraries are needed for the C167/ST10x167/ST10x262 and similar architectures. These architectures feature the extended instruction set, extended special function registers, 24–bit addressing and extended PEC pointers. Use these libraries in conjunction with the compiler option **–x**.
extp	The protected libraries provide a software workaround for CPU functional problems. Use these libraries in conjunction with the compiler options **–x** and **–B**.
ext2	The extended 2 libraries are needed for the C166S v2.0 / Super10 and similar architectures. These architectures feature jump prediction, scalable and relocatable interrupt vector table, local register banks and instruction reordering. Use these libraries in conjunction with the compiler option **–x2**.
ext2p	The protected libraries provide a software workaround for CPU functional problems. Use these libraries in conjunction with the compiler options **–x2** and **–B**.

Another seven sets of libraries are delivered to meet specific User Stack Model requirements for the various microcontroller architectures. These libraries must be used in conjunction with the additional compiler option –P. These sets are located in separate directories:

u166 The User Stack Model variant of the non–protected libraries.

u166p The User Stack Model variant of the protected libraries.

ugoldp The User Stack Model variant of the protected Gold architecture libraries.

uext The User Stack Model variant of the extended non–protected libraries.

uextp The User Stack Model variant of the extended protected libraries.

uext2 The User Stack Model variant of the extended C166S v2.0 / Super10 architectures non–protected libraries.

uext2p The User Stack Model variant of the extended C166S v2.0 / Super10 architectures protected libraries.

Each library set contains the following libraries:

c166?[s].lib C library. The optional [s] stands for single precision floating point (all floating point arithmetic is in single precision instead of ANSI double precision).

fp166?[t].lib
Floating point library. The optional [t] stands for trapping floating point (using boundary checking and the floating point trap mechanism).

rt166?[s][m].lib
Run–time library. The optional [s] stands for single precision floating point. The optional [m] stands for MAC optimized (use MAC instructions in some basic operations for optimization).

The question mark '?' in these library names must be replaced by a letter representing the selected memory model:

t tiny
s small
m medium
l large

All C library functions are described in the section *C Library Interface Description*. These functions are only called by explicit function calls in your application program. However, some compiler generated code contain calls to run–time library functions that would use too much code when generated as inline code. The name of a run–time library function always contains two leading underscores. For example, to perform a long (32 bit) signed division, the function `__sdil` is called.

Because **c166** generates assembly code (and not object code) it adds a leading underscore to the names of (public) C variables to distinguish these symbols from 80166 registers. So if you use a function with a leading underscore, the assembly label for this function contains two leading underscores. This function name could cause a name conflict (double defined) with one of the run–time library functions. Therefore, you should avoid names starting with an underscore. Note that ANSI states that it is not portable to use names starting with an underscore for public C variables and functions, because results are implementation defined.

The code sections of the C166 library have the class 'CLIBRARY', 'SHAREDCLIB', 'RTLIBRARY' or 'SHAREDRTLIB' allowing the library to be allocated in a special memory area via the CLASSES control of **l166**.

6.2 SMALL, MEDIUM AND LARGE I/O FORMATTERS

The C library contains the SMALL I/O formatter version of the printf() and scanf() functions and their variants like sprintf(), fprintf(), etc. This SMALL version does not contain the required functionality to handle precision specifiers and floating point I/O which can specified in the format argument of these functions.

The following extra libraries are included to support easy switching between the three I/O formatter versions:

MEDIUM I/O formatter library no floating point I/O supported precision specifiers supported fmtio?m.lib.

LARGE I/O formatter library floating point I/O supported precision specifiers supported fmtio?l[s].lib.

The question mark '?' in these library names must be replaced by a character representing the selected memory model:

t tiny
s small
m medium
l large

These I/O formatter libraries are included in all library sets. The control program options **–libfmtiom** and **–libfmtiol** can be used to selected the MEDIUM and LARGE I/O formatter libraries.

If no **cc166 –libfmtio*** option is specified on the commandline, then the SMALL printf()/scanf() formatter variant is linked from the C library.

6.3 SINGLE PRECISION FLOATING POINT

In ANSI C all mathematical functions (`<math.h>`), are based on `double` arguments and `double` return type. So, even if you are using only `float` variables in your code, the language definition dictates promotion to `double`, when using the math functions or floating point formatters (`printf()` and `scanf()`). The result is more code and less execution speed. In fact the ANSI approach introduces a performance penalty.

To improve the code size and execution speed, the compiler now supports the option **–F** to force single precision floating point usage. If you use **–F**, a `float` variable passed as an argument is no longer promoted to `double` when calling a variable argument function or an old style K&R function, and the type `double` is treated as `float`. It is obvious that this affects the whole application (including libraries). Therefore special single precision versions of the floating point libraries are now delivered with the package. When using **–F**, these libraries must be used. It is not possible to mix C modules created with the **–F** option and C modules which are using the regular ANSI approach.

For compatibility with the old **–F** option, the **–Fc** option is introduced. This option only treats floating point constants (having no suffix) as `float` instead of `double`.