

CROSSWARE⁷ COLDFIRE

C/C++ Compiler

The Crossware ColdFire C/C++ compiler generates code for the Freescale ColdFire MCF52xx, MCF53xx and MCF54xx families of microprocessors and microcontrollers. It comes as part of the Crossware ColdFire Development Suite and runs under Windows 9x, Windows NT 4.0, Windows 2000 and Windows XP.

HIGHLIGHTS

- Optimising C/C++ compiler with extensions for embedded development.
- Pre-written library routines including 32-bit and 64-bit floating point arithmetic.
- Support for 64-bit integer arithmetic
- Comprehensive source level debug output.
- Data output for Embedded Development Studio browser.

C Language Definition

The C compiler conforms to the 1989 ANSI C specification and in addition provides a number of general enhancements including:

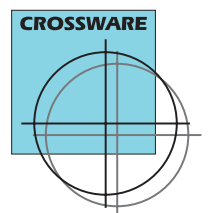
- variables can be any length with all characters significant
- the `_interrupt` keyword declares a C function as an interrupt routine
- the `_persist` keyword declares that a variable will retain its contents (eg. in battery backed up ram) during power down and should therefore not be initialized.

It also supports a number of features from the 1999 ANSI C Standard including:

- The `//` characters mark the start of a comment which extends to the end of the line.
- Variables can be defined anywhere within a block, not just at the beginning.
- Variables can be defined within the initialising expression of a **for** loop.

In addition, the compiler supports the C++ feature that allows variables to be defined within the conditional expressions of the **for** loop, **while** loop, **if** statement and **switch** statement

The support libraries are a subset of the ANSI standard libraries. The supported functions are listed below.



*Crossware Products
Old Post House
Silver Street
Litlington
Royston
Herts
SG8 0QE
UK*

*Telephone
+ 44 (0) 1763 853500*

*Facsimile
+ 44 (0) 1763 853330*

*Web
<http://www.crossware.com>*

*E-mail
info@crossware.com*

C++ Language Definition

The C++ compiler supports the Embedded C++ language. Embedded C++ is a subset of C++ specifically formulated for embedded systems. It is defined at <http://www.caravan.net/ec2plus>.

Embedded C++ excludes templates, exceptions, namespaces, run-time type information, localization, file operations and some other features.

Also Embedded C++ does not support multiple-inheritance and virtual base classes. However the Crossware compiler does support multiple-inheritance but does not support virtual base classes.

Since Embedded C++ does not support templates, `basic_string` is not supported. As an alternative, Embedded C++ provides the `string` class. The Crossware C++ library includes the `string` class.

The Crossware C++ library also includes the operators `new`, `delete`, `new[]`, `delete[]`, `placement new` and `placement new[]`.

Other Embedded C++ library features such as streams and complex are not yet available in the Crossware library.

Data Sizes

The compiler uses the following sizes for the various C data types:

char and unsigned char	1 byte
short int and unsigned short int	2 bytes
int and unsigned int	4 bytes
long and unsigned long	4 bytes
long long and unsigned long long	8 bytes
float	4bytes (32 bits)
double	8 bytes (64 bits)
long double	8 bytes (64 bits)
enum	up to 4 bytes (minimum size to accommodate members)
bit fields	up to 32 bits

Optimizations

Optimizations include:

- constant folding
- dead code elimination
- strength reduction
- algebraic simplification
- jump/branch optimization
- suppression of integral promotion
- global register allocation

Since the introduction of C compiler version 3 of the C compiler, additional optimizations can be enabled by instructing the compiler to perform a data flow analysis. Enabling this 'advanced features' option leads to:

- register allocation by graph coloring
- optional function in-lining
- the availability of additional function calling conventions
- additional optimizations enabled by information from the data flow analysis

The 'advanced features' option is always enabled for C++ source code.

Function Calling Conventions

When the 'advanced features' option is enabled, the compiler supports three calling conventions:

1. Arguments declared with the register keyword will be passed in registers if an appropriate register is available. Integer arguments are passed in registers D1 and D0. Pointer arguments are passed in registers A1 and A0. If the chip has a floating point unit, floating point arguments are passed in floating point registers FP1 and FP0. The arguments are popped from the stack after the called function returns. (Caller pops stack.)
2. As above except that arguments are popped from the stack by the called function before it returns. (Callee pops stack.)
3. If an appropriate register is available, an argument will be passed in a register. Integer arguments are passed in registers D1 and D0. Pointer arguments are passed in registers A1 and A0. If the chip has a floating point unit, floating point arguments are passed in floating point registers FP1 and FP0. The register keyword will be ignored. The arguments are popped from the stack by the called function before it returns. (Callee pops stack.)

In-Line Assembler

Assembler code can be embedded in your C source code using two methods.

The `_asm` keyword can be used to embed assembler into C functions. The `#asm/#endasm` directive allows assembler to be placed anywhere within a C source file, not just within functions.

Strings inserted using the `_asm` keyword are scanned by the C preprocessor and so assembler statements can be generated using C macros with full macro token replacement. Additionally, the compiler can replace C variable names with the appropriate substring allowing easy access to global, static and local variables and parameters.

On the other hand, assembler macros are best defined outside of functions and the `#asm/#endasm` directive allows this.

```
#define enable_interrupts( level ) \
    _asm(" move.w ({\" #level \"},A6,d0"); \
    _asm(" move.w d0,SR");

#define disable_interrupts( level ) \
    _asm(" move.w SR,d0"); \
    _asm(" move.w d0,({\" #level \"},A6)"); \
    _asm(" move.l #$0700,d1"); \
    _asm(" or.l d1,d0"); \
    _asm(" move.w d0,SR");

int gettimeofday(struct timeval *tp)
{
    unsigned long level;
    unsigned long seconds;
    unsigned long microseconds;
    if (tp == NULL)
        return -1;
    disable_interrupts(level);
    seconds = Seconds_since_epoch;
    microseconds = Current_ticks;
    enable_interrupts(level);
    tp->tv_sec = seconds + SECONDS_ADJUST;
    tp->tv_usec = microseconds * Microseconds_per_tick;
    return 0;
}
```

Code and data location

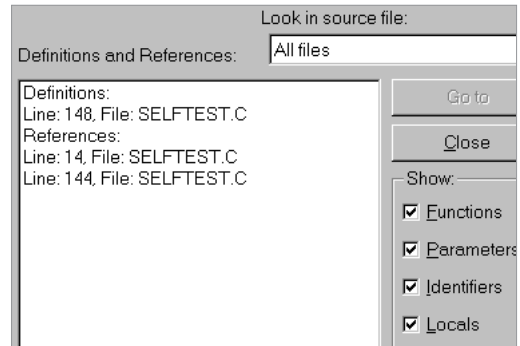
Compiler generated code and data are automatically located in appropriate memory segments. Linker options allow these segments to be located at user defined memory locations. String constants and objects declared as `const` are located in code space. Initialised and uninitialised data are located in separate segments and initialisation of these segments is automatically carried out at run time.

Directives are also available allowing you to tell the compiler to place code and data in your own named segments. This gives you further control on where code and data are placed and how data space is initialised.

The Diab Data **section** and **use_section** pragmas are also supported allowing code that uses these to be compiled unchanged.

Source Code Browsing

The compiler optionally generates information on all of the definitions of and references to the identifiers used in your program. This includes functions, function parameters, local variables, global and static variables, enum identifiers, typedefs, goto labels and the tag names of structures, unions and enums. The Embedded Development Studio will then use this information to allow you to quickly navigate through your source code.



Debug Records

Comprehensive debug records are generated by the compiler and embedded in the object files for each module. These are output by the linker to the final program file.

C Library Routines:

abs()	cosh()	gets()	longjmp()	sin()	strchr()
acos()	exit()	getvect()	ltoa()	sinl()	strspn()
acosh()	exp()	gmtime()	malloc()	sinh()	strstr()
alloca()	expl()	isalnum()	memchr()	sinhl()	tan()
asctime()	fabs()	isalpha()	memcmp()	sprintf()	tanl()
asin()	fabsl()	isascii()	memcpy()	sqrt()	tanh()
asinl()	_fcvt()	isctrl()	memmove()	sqrtl()	tanhf()
atan()	ferroe()	isdigit()	memset()	srand()	time()
atanl()	fgetc()	isgraph()	mktime()	sscanf()	toascii()
atoi()	fgets()	islower()	pow()	strcat()	tolower()
atol()	fileno()	isprint()	powl()	sscanf()	toupper()
atoff()	floor()	ispunct()	printf()	strchr()	ultoa()
atolf()	floorl()	isspace()	putc()	strcmp()	ungetc()
calloc()	fprintf()	isupper()	putchar()	strcpy()	ungetch()
clearerr()	fputc()	isxdigit()	puts()	strcspn()	vfprintf()
ceil()	fputs()	labs()	qsort()	stricmp()	vprintf()
ceilf()	free()	ldexp()	rand()	strlen()	vsprintf()
clock()	fscanf()	log()	sbrk()	strncat()	
cos()	getc()	logf()	scanf()	strncmp()	
cosl()	getchar()	log10()	setjmp()	strncpy()	
cosh()	getche()	log10l()	setvect()	strpbrk()	