



Version 4.1

***C Cross Compiler User's Guide
for Freescale XGATE***

Copyright © COSMIC Software 1995, 2005
All Trademarks are the property of their respective owners

Table of Contents

Preface

Organization of this Manual	1
-----------------------------------	---

Chapter 1

Introduction

Introduction.....	4
Document Conventions.....	4
Typewriter font.....	4
Italics	5
[Brackets]	5
Conventions.....	6
Command Line	6
Flags	6
Compiler Architecture	8
Predefined Symbol.....	9
Linking.....	9
Programming Support Utilities.....	9
Listings.....	9
Optimizations.....	10

Chapter 2

Tutorial Introduction

Xtest.c, Example file.....	14
Default Compiler Operation	15
Compiling and Linking.....	15
Step 1: Compiling.....	15
Step 2: Assembling.....	17
Step 3: Linking	17
Automatic Code and Data Initialization	20
Specifying Command Line Options	20

Chapter 3

Programming Environments

Introduction.....	24
The const and volatile Type Qualifiers.....	25
Performing Input/Output in C.....	26
Placing Data Objects in The Bss Section.....	27
Redefining Sections	27
Referencing Absolute Addresses	28
Accessing Internal Registers.....	29

Inserting Inline Assembly Instructions.....	30
Inlining with pragmas.....	30
Inlining with <code>_asm</code>	31
Inlining Labels.....	33
Writing Interrupt Handlers	34
Placing Addresses in Gate Vectors.....	35
Inline Function.....	36
Interfacing C to Assembly Language	38
Register Usage.....	39
Stack Display.....	39
Data Representation.....	42

Chapter 4

Using The Compiler

Invoking the Compiler.....	44
Compiler Command Line Options	45
File Naming Conventions	49
Generating Listings.....	50
Generating an Error File	50
Return Status.....	50
Examples	50
C Library Support.....	51
How C Library Functions are Packaged.....	51
Inserting Assembler Code Directly	51
Linking Libraries with Your Program.....	51
Integer Library Functions.....	51
Common Input/Output Functions.....	52
Functions Implemented as Macros.....	52
Including Header Files	52
Descriptions of C Library Functions	54
Generate inline assembly code.....	55
Abort program execution.....	56
Find absolute value.....	57
Arccosine.....	58
Arcsine.....	59
Arctangent	60
Arctangent of y/x	61
Convert buffer to double	62
Convert buffer to integer	63
Convert buffer to long	64
Get the first bit set position	65
Allocate and clear space on the heap.....	66

Test or get the carry bit.....	67
Round to next higher integer.....	68
Cosine.....	69
Hyperbolic cosine.....	70
Clear a semaphore.....	71
Divide with quotient and remainder.....	72
Exit program execution.....	73
Exponential.....	74
Find double absolute value.....	75
Round to next lower integer.....	76
Find double modulus.....	77
Free space on the heap.....	78
Extract fraction from exponent part.....	79
Get character from input stream.....	80
Get a text line from input stream.....	81
Test for alphabetic or numeric character.....	82
Test for alphabetic character.....	83
Test for control character.....	84
Test for digit.....	85
Test for graphic character.....	86
Test for lowercase character.....	87
Test for printing character.....	88
Test for punctuation character.....	89
Integer square root.....	90
Test for whitespace character.....	91
Test for uppercase character.....	92
Test for hexadecimal digit.....	93
Find long absolute value.....	94
Scale double exponent.....	95
Long divide with quotient and remainder.....	96
Natural logarithm.....	97
Common logarithm.....	98
Restore calling environment.....	99
Long integer square root.....	100
Allocate space on the heap.....	101
Test for maximum.....	102
Scan buffer for character.....	103
Compare two buffers for lexical order.....	104
Copy one buffer to another.....	105
Copy one buffer to another.....	106
Propagate fill character throughout buffer.....	107
Test for minimum.....	108

Extract fraction and integer from double	109
Test or get the parity.....	110
Raise x to the y power	111
Output formatted arguments to stdout.....	112
Put a character to output stream	117
Put a text line to output stream.....	118
Generate pseudo-random number	119
Reallocate space on the heap.....	120
Allocate new memory	121
Read formatted input.....	122
Save calling environment	126
Set interrupt flag	128
Sin.....	129
Hyperbolic sine.....	130
Output arguments formatted to buffer.....	131
Real square root.....	132
Seed pseudo-random number generator	133
Read formatted input from a string	134
Set a semaphore.....	135
Concatenate strings.....	136
Scan string for first occurrence of character	137
Compare two strings for lexical order.....	138
Copy one string to another	139
Find the end of a span of characters in a set.....	140
Find length of a string	141
Concatenate strings of length n	142
Compare two n length strings for lexical order.....	143
Copy n length string	144
Find occurrence in string of character in set	145
Scan string for last occurrence of character	146
Find the end of a span of characters not in set	147
Scan string for first occurrence of string	148
Convert buffer to double	149
Convert buffer to long	150
Convert buffer to unsigned long.....	151
Tangent.....	152
Hyperbolic tangent	153
Convert character to lowercase if necessary	154
Convert character to uppercase if necessary	155
Get pointer to next argument in list.....	156
Stop accessing values in an argument list	158
Start accessing values in an argument list.....	160

Output arguments formatted to stdout.....	162
Output arguments formatted to buffer.....	163

Chapter 5

Using The Assembler

Invoking caxgate.....	166
Object File.....	168
Listings.....	169
Assembly Language Syntax.....	170
Instructions	170
Labels	171
Temporary Labels.....	172
Constants	172
Expressions	173
Macro Instructions.....	175
Conditional Directives.....	178
Sections.....	179
Includes.....	179
Branch Optimization.....	180
C Style Directives	181
Assembler Directives.....	181
Align the next instruction on a given boundary	182
Define the default base for numerical constants.....	183
Switch to the predefined .bsct section.	184
Turn listing of conditionally excluded code on or off.	185
Allocate constant(s)	186
Allocate constant block	187
Turn listing of debug directives on or off.....	188
Allocate variable(s)	189
Conditional assembly	190
Conditional assembly	191
Stop the assembly	192
End conditional assembly.....	193
End conditional assembly.....	194
End macro definition	195
End repeat section.....	196
Give a permanent value to a symbol	197
Assemble next byte at the next even address relative to the start of a section.....	198
Generate error message.	199
Conditional assembly	200
Conditional assembly	201

Conditional assembly	202
Conditional assembly	203
Conditional assembly	204
Conditional assembly	205
Conditional assembly	206
Conditional assembly	207
Conditional assembly	208
Conditional assembly	209
Conditional assembly	210
Include text from another text file	211
Turn on listing during assembly	212
Give a text equivalent to a symbol	213
Create a new local block	214
Define a macro	215
Send a message out to STDOUT	217
Terminate a macro definition	218
Turn on or off listing of macro expansion	219
Turn off listing	220
Disable pagination in the listing file	221
Creates absolute symbols	222
Sets the location counter to an offset from the beginning of a section	223
Start a new page in the listing file	224
Specify the number of lines per pages in the listing file ..	225
Repeat a list of lines a number of times	226
Repeat a list of lines a number of times	227
Restore saved section	229
Terminate a repeat definition	230
Save section	231
Define a new section	232
Give a resetable value to a symbol	234
Insert a number of blank lines before the next statement in the listing file	235
Place code into a section.	236
Specify the number of spaces for a tab character in the listing file	237
Define default header	238
Declare a variable to be visible	239
Declare symbol as being defined elsewhere	240

Chapter 6	
Using The Linker	
Linking XGATE Objects	242
Linking Library Objects.....	243

Chapter 7

Debugging Support

Chapter 8

Programming Support

Chapter A

Compiler Error Messages

Parser (cpxgate) Error Messages	250
Code Generator (cgxgate) Error Messages.....	264
Assembler (caxgate) Error Messages	265
Linker (clnk) Error Messages	268

Chapter B

Modifying Compiler Operation

The Configuration File.....	272
Changing the Default Options	273
Creating Your Own Options.....	273
Example	274

Chapter C

XGATE Machine Library

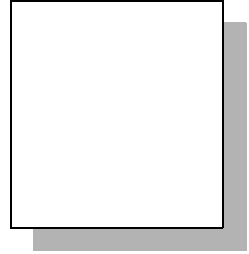
Function Listing	275
------------------------	-----

Chapter D

Compiler Passes

The cpxgate Parser	278
Command Line Options	278
Return Status	281
Example	281
The cgxgate Code Generator	283
Command Line Options	283
Return Status	284
Example	284
The coxgate Assembly Language Optimizer.....	286
Command Line Options	286

Disabling Optimization	287
Return Status	287
Example.....	287



Preface

The *Cross Compiler User's Guide for [XGATE](#)* is a reference guide for programmers writing C programs for [XGATE](#) microcontroller environments. It provides an overview of how the cross compiler works, and explains how to compile, assemble, link and debug programs. It also describes the programming support utilities included with the cross compiler and provides tutorial and reference information to help you configure executable images to meet specific requirements. This manual assumes that you are familiar with your host operating system and with your specific target environment.

Organization of this Manual

This manual is divided into eight chapters and four appendixes.

Chapter 1, “[Introduction](#)”, describes the basic organization of the C compiler and programming support utilities.

Chapter 2, “[Tutorial Introduction](#)”, is a series of examples that demonstrates how to compile, assemble and link a simple C program.

Chapter 3, “[Programming Environments](#)”, explains how to use the features of C for [XGATE](#) to meet the requirements of your particular application. It explains how to create a runtime startup for your application, and how to write C routines that perform special tasks such as: serial I/O, direct references to hardware addresses, interrupt handling, and assembly language calls.

Chapter 4, “[Using The Compiler](#)”, describes the compiler options. This chapter also describes the functions in the C runtime library.

Chapter 5, “[Using The Assembler](#)”, describes the **XGATE** assembler and its options. It explains the rules that your assembly language source must follow, and it documents all the directives supported by the assembler.

Chapter 6, “[Using The Linker](#)”, describes the linker and its options. This chapter describes in detail all the features of the linker and their use. As the XGATE compiler is an add-on to the S12X compiler, the linker is not provided. You should then read the COSMIC’s *S12X Cross compiler User’s Guide*.

Chapter 7, “[Debugging Support](#)”, describes the support available for COSMIC’s C source level cross debugger and for other debuggers or in-circuit emulators. As the XGATE compiler is an add-on to the S12X compiler, the debugging support utilities are not provided. You should then read the COSMIC’s *S12X Cross compiler User’s Guide*.

Chapter 8, “[Programming Support](#)”, describes the programming support utilities. Examples of how to use these utilities are also included. As the XGATE compiler is an add-on to the S12X compiler, the programming support utilities are not provided. You should then read the COSMIC’s *S12X Cross compiler User’s Guide*.

Appendix A, “[Compiler Error Messages](#)”, is a list of compile time error messages that the C compiler may generate.

Appendix B, “[Modifying Compiler Operation](#)”, describes the “configuration file” that serves as default behaviour to the C compiler.

Appendix C, “[XGATE Machine Library](#)”, describes the assembly language routines that provide support for the C runtime library.

Appendix D, “[Compiler Passes](#)”, describes the specifics of the parser, code generator and assembly language optimizer and the command line options that each accepts.

This manual also contains an Index.

CHAPTER 1

Introduction

This chapter explains how the compiler operates. It also provides a basic understanding of the compiler architecture. This chapter includes the following sections:

- Introduction
- Document Conventions
- Compiler Architecture
- Predefined Symbol
- Linking
- Programming Support Utilities
- Listings
- Optimizations

Introduction

The C cross compiler targeting the XGATE microcontroller reads C source files, assembly language source files, and object code files, and produces an executable file. You can request listings that show your C source interspersed with the assembly language code and object code that the compiler generates. You can also request that the compiler generate an object module that contains debugging information that can be used by COSMIC's C source level cross debugger or by other debuggers or in-circuit emulators.

You begin compilation by invoking the **cxxgate** compiler driver with the specific options you need and the files to be compiled.

Document Conventions

In this documentation set, we use a number of styles and typefaces to demonstrate the syntax of various commands and to show sample text you might type at a terminal or observe in a file. The following is a list of these conventions.

Typewriter font

Used for user input/screen output. Typewriter (or courier) font is used in the text and in examples to represent what you might type at a terminal: command names, directives, switches, literal filenames, or any other text which must be typed exactly as shown. It is also used in other examples to represent what you might see on a screen or in a printed listing and to denote executables.

To distinguish it from other examples or listings, input from the user will appear in a shaded box throughout the text. Output to the terminal or to a file will appear in a line box.

For example, if you were instructed to type the compiler command that generates debugging information, it would appear as:

```
cxxgate +debug acia.c
```

Typewriter font enclosed in a shaded box indicates that this line is entered by the user at the terminal.

If, however, the text included a partial listing of the file *acia.c* ‘an example of text from a file or from output to the terminal’ then typewriter font would still be used, but would be enclosed in a line box:

```
/* defines the ACIA as a structure */
struct acia {
    char status;
    char data;
} acia @0x6000;
```

NOTE

Due to the page width limitations of this manual, a single invocation line may be represented as two or more lines. You should, however, type the invocation as one line unless otherwise directed.

Italics

Used for value substitution. *Italic* type indicates categories of items for which you must substitute appropriate values, such as arguments or hypothetical filenames. For example, if the text was demonstrating a hypothetical command line to compile and generate debugging information for any file, it might appear as:

```
cxxgate +debug file.c
```

In this example, `cxxgate +debug file.c` is shown in typewriter font because it must be typed exactly as shown. Because the filename must be specified by the user, however, *file* is shown in italics.

[Brackets]

Items enclosed in brackets are optional. For example, the line:

[*options*]

means that zero or more options may be specified because options appears in brackets. Conversely, the line:

options

means that one or more options must be specified because options is not enclosed by brackets.

As another example, the line:

```
file1.[o|xgt]
```

means that one file with the extension `.o` or `.xgt` may be specified, and the line:

```
file1 [ file2 . . . ]
```

means that additional files may be specified.

Conventions

All the compiler utilities share the same optional arguments syntax. They are invoked by typing a command line.

Command Line

A command line is generally composed of three major parts:

```
program_name [<flags>] <files>
```

where *<program_name>* is the name of the program to run, *<flags>* an optional series of flags, and *<files>* a series of files. Each element of a command line is usually a string separated by whitespace from all the others.

Flags

Flags are used to select options or specify parameters. Options are recognized by their first character, which is always a `'-'` or a `'+'`, followed by the name of the flag (usually a single letter). Some flags are simply *yes* or *no* indicators, but some must be followed by a value or some additional information. The value, if required, may be a character string, a single character, or an integer. The flags may be given in any order, and two or more may be combined in the same argument, so long as the second flag can't be mistaken for a value that goes with the previous one.

It is possible for each utility to display a list of accepted options by specifying the **-help** option. Each option will be displayed alphabetically on a separate line with its name and a brief description. If an option requires additional information, then the type of information is

indicated by one of the following code, displayed immediately after the option name:

Code	Type of information
*	character string
#	short integer
##	long integer
?	single character

If the code is immediately followed by the character '>', the option may be specified more than once with different values. In that case, the option name must be repeated for every specification.

For example, the options of the **chex** utility are:

```
chex [options] file
-a##      absolute file start address
-b##      address bias
-e##      entry point address
-f?       output format
-h        suppress header
+h*       specify header string
-m#       maximum data bytes per line
-n*>      output only named segments
-o*       output file name
-p        use paged address format
-pp       use paged address with mapping
-pn       use paged address in bank only
-s        output increasing addresses
-x*       exclude named segment
```

chex accepts the following distinct flags:

Flags	Function
-a	accept a long integer value
-b	accept a long integer value
-e	accept a long integer value
-f	accept a single character
-h	simply a flag indicator
+h	accept a character string
-m	accept a short integer value,
-n	accept a character string and may be repeated
-o	accept a character string
-p	simply a flag indicator
-pn	simply a flag indicator
-pp	simply a flag indicator
-s	simply a flag indicator
-x	accept a character string and may be repeated

Compiler Architecture

The C compiler consists of several programs that work together to translate your C source files to executable files and listings. **cxsgate** controls the operation of these programs automatically, using the options you specify, and runs the programs described below in the order listed:

cpsgate - the C preprocessor and language parser. *cpsgate* expands directives in your C source and parses the resulting text.

cgsgate - the code generator. *cgsgate* accepts the output of *cpsgate* and generates assembly language statements.

coxgate - the assembly language optimizer. *coxgate* optimizes the assembly language code that *cgsgate* generates.

caxgate - the assembler. *caxgate* converts the assembly language output of *coxgate* to a relocatable object module.

Predefined Symbol

The COSMIC compiler defines the `__CSMC__` preprocessor symbol. It expands to a numerical value whose each bit indicates if a specific option has been activated:

- bit 2: set if unsigned char option specified (**-pu**)
- bit 4: set if reverse bitfield option specified (**+rev**)
- bit 5: set if no enum optimization specified (**-pne**)

Linking

clnk combines all the object modules that make up your program with the appropriate modules from the C library. You can also build your own libraries and have the linker select files from them as well. The linker generates an executable file which, after further processing with the *chex* utility, can be downloaded and run on your target system. If you specify debugging options when you invoke **cxgate**, the compiler will generate a file that contains debugging information. You can then use the COSMIC's debugger to debug your code.

Programming Support Utilities

Once object files are produced, you run **clnk** (the linker) to produce an executable image for your target system; you can use the programming support utilities to inspect the executable. As the XGATE compiler is an add-on to the S12X compiler, the programming support utilities are not provided. You should then read the *COSMIC's S12X C Cross compiler Users' Guide*.

Listings

Several options for listings are available. If you request no listings, then error messages from the compiler are directed to your terminal, but no additional information is provided. Each error is labelled with the C source file name and line number where the error was detected.

If you request an assembly language and object code listing with interspersed C source, the compiler merges the C source as comments among the assembly language statements and lines of object code that it generates. Unless you specify otherwise, the error messages are still written to your terminal. Your listing is the listing output from the assembler.

Optimizations

The C cross compiler performs a number of compile time and optimizations that help make your application smaller and faster:

- The compiler will perform arithmetic operations in 8-bit precision if the operands are 8-bit.
- The compiler eliminates unreachable code.
- Branch shortening logic chooses the smallest possible jump/branch instructions. Jumps to jumps and jumps over jumps are eliminated as well.
- Integer and float constant expressions are folded at compile time.
- Redundant load and store operations are removed.
- **enum** is large enough to represent all of its declared values, each of which is given a name. The names of **enum** values occupy the same space as type definitions, functions and object names. The compiler provides the ability to declare an **enum** using the smallest type char, int or long:
- The compiler performs multiplication by powers of two as faster shift instructions.
- An optimized switch statement produces combinations of tests and branches, jump tables for closely spaced case labels, a scan table for a small group of loosely spaced case labels, or a sorted table for an efficient search.
- The functions in the C library are packaged in three separate libraries; one of them is built without floating point support. If

your application does not perform floating point calculations, you can decrease its size and increase its runtime efficiency by linking with the non-floating-point version of the modules needed.

For information on using the compiler, see [Chapter 4](#).

For information on using the assembler, see [Chapter 5](#).

For information on using the linker, see [Chapter 6](#) and the [Chapter 6](#) of the *S12X Cross Compiler User's Guide*.

For information on debugging support, see the [Chapter 7](#) of the *S12X Cross Compiler User's Guide*.

For information on using the programming utilities, see the [Chapter 8](#) of the *S12X Cross Compiler User's Guide*.

For information on the compiler passes, see [Appendix D](#).

CHAPTER 2

Tutorial Introduction

This chapter will demonstrate, step by step, how to compile, assemble and link the example program `xtest.c`, which is included on your distribution media. Although this tutorial cannot show all the topics relevant to the COSMIC tools, it will demonstrate the basics of using the compiler for the most common applications.

The code produced by the XGATE compiler will be linked with a hosting S12X application and cannot produced stand alone code. Examples of link command files are sub parts of complete S12X linker files.

In this tutorial you will find information on the following topics:

- [Default Compiler Operation](#)
- [Compiling and Linking](#)
- [Automatic Code and Data Initialization](#)
- [Specifying Command Line Options](#)

Xtest.c, Example file

The following is a listing of *xtest.c*. This C source file is copied during the installation of the compiler:

```

/*      EXAMPLE PROGRAM FOR XGATE CODE
 *      Copyright (c) 2005 by COSMIC Software
 */
#include <ioxdp512.h>
#include <processor.h>

/*      dummy function for vector table
 */
@interrupt void xdumint(void)
{
}

/*      function started by trigger 0
 *      summing the elements from the array
 */
@interrupt void xsum(int *tab, int nb, int ares)
{
    int res = ares;

    res = 0;                // clear result
    do {
        res += *tab++;      // sum element
    } while (--nb);        // count down
    XGSWT = 0x100;         // acknowledge trigger
    _sif();                // send S12X interrupt
}

extern int tab[];          // S12X array address

/*      gate for xsum function
 *      using static initialization
 */
struct argument {
    int *tab;              // array address
    int nb;               // number of elements
    int res;              // result value
} xargsum = {tab, 10, 0};

```


Default Compiler Operation

By default, the compiler compiles and assembles your program. You may then link object files using **clnk** to create an executable program.

As it processes the command line, **cxxgate** echoes the name of each input file to the standard output file (your terminal screen by default). You can change the amount of information the compiler sends to your terminal screen using command line options, as described later.

According to the options you will use, the following files, recognized by the COSMIC naming conventions, will be generated:

file.s	Assembler source module
file.o	Relocatable object module
file.xgt	input (<i>e.g.</i> libraries) file for the linker

Compiling and Linking

To compile and assemble *xtest.c* using default options, type:

```
cxxgate xtest.c
```

The compiler writes the name of the input file it processes:

```
xtest.c:
```

The result of the compilation process is an object module named *xtest.o* produced by the assembler. We will, now, show you how to use the different components.

Step 1: Compiling

The first step consists in compiling the C source file and producing an assembly language file named **xtest.s**.

```
cxxgate -s xtest.c
```

The **-s** option directs **cxxgate** to stop after having produced the assembly file *xtest.s*. You can then edit this file with your favorite editor. You

can also visualize it with the appropriate system command (*type*, *cat*, *more*,...). For example under MS/DOS you would type:

```
type xtest.s
```

If you wish to get an interspersed C and assembly language file, you should type:

```
cxxgate -l xtest.c
```

The **-l** option directs the compiler to produce an assembly language file with C source line interspersed in it. Please note that the C source lines are commented in the assembly language file: they start with ‘;’.

As you use the C compiler, you may find it useful to see the various actions taken by the compiler and to verify the options you selected. The **-v** option, known as verbose mode, instructs the C compiler to display all of its actions. For example if you type:

```
cxxgate -v -s xtest.c
```

the display will look like something similar to the following:

```
xtest.c:
  cpxgate -o \2.cx1 -i\cx12x\hxgate -u xtest.c
  cgxgate -o \2.cx2 \2.cx1
  coxgate -o xtest.s \2.cx2
```

The compiler runs each pass:

cpxgate	the C parser
cgxgate	the assembly code generator
coxgate	the optimizer

For more information, see **Appendix D**, “[Compiler Passes](#)”.

Step 2: Assembling

The second step of the compilation is to assemble the code previously produced. The relocatable object file produced is *xtest.o*.

```
cxxgate xtest.s
```

or

```
caxgate -i\cx12x\hxxgate xtest.s
```

if you want to use directly the macro cross assembler.

The cross assembler can provide, when necessary, listings, symbol table, cross reference and more. The following command will generate a listing file named *xtest.ls* that will also contain a cross reference:

```
caxgate -c -l xtest.s
```

For more information, see **Chapter 5**, “[Using The Assembler](#)”.

Step 3: Linking

This step consists in linking relocatable files, also referred to as object modules, produced by the compiler or by the assembler (<files>.o) and object files produced by the COSMIC S12X compiler into an absolute executable file: *xtest.x12* in our example. Code and data sections will be located at absolute memory addresses. The S12X linker is used with a command file (*xtest.lkf* in this example).

An application that uses one or more object module(s) may require several sections (code, data, gate vectors, etc.,...) located at different addresses. Each object module contains several sections. The compiler creates the following sections:

Type	Description
.xtext	code (or program) section
.xconst	constant and literal data

Type	Description
.xdata	initialized data
.xbss	non initialized data

In our example, and in the test file provided with the compiler, the *xtest.lkf* file contains the following information:

```

line 1 # LINK COMMAND FILE FOR TEST PROGRAM
line 2 # Copyright (c) 2005 by COSMIC Software
line 3 #
line 4 +seg .xtext -b0xfb000 -pr -id -n.xtext # program start
line 5 +seg .xconst -a .xtext -pr -id # constants follow code
line 6 +seg .xdata -b0xfd000 -o0x1000 -pr -id -n.xdata # data start
line 7 +seg .xbss -a .xdata -pr # constants follow code
line 8 xtest.o # application program
line 9 +pri
line 10 \cx\lib\libi.xgt # C library (if needed)
line 11 +new
line 12 \cx\lib\libm.xgt # machine library
line 13 +def __xstack=0xdffe # xgate stack pointer value

```

You can create your own link command file by modifying the one provided with the compiler.

Here is the explanation of the lines in *xtest.lkf*:

lines 1 to 3: These are comment lines. Each line can include comments. They must be prefixed by the “#” character.

line 4: `+seg .xtext -b0xfb000 -pr -id` creates a text (code) segment located at global address **fb000** (hexa) matching XGATE address **b000**. The **-pr** option enforces physical relocation while the **-id** option allows the segment to be copied in RAM at startup.

line 5: `+seg .xconst -a.xtext -pr -id` creates a constant segment located after the previous xtext segment.

line 6: `+seg .xdata -b0xfd000 -o0x1000 -pr -id` creates a data segment located at global address **fd100** matching XGATE address **d000** when linking an XGATE segment, and S12X address **1000** when linking a S12X segment.

line 7: `+seg .xbss -a.xdata -pr` creates a data segment located after the previous xdata segment. No `-id` option is specified as this segment contains an uninitialized data.

line 8: `xtest.o`, the file that constitutes your application. It follows the startup routine for code and data.

line 9: `+pri` starts a private region in order to avoid symbol redefinition between XGATE and S12X standard library functions.

line 10: `libi.xgt` the integer library to resolve references

line 11: `+new` restarts a public region.

line 12: `libm.xgt` the machine library to resolve references

line 13: `+def __xstack=0xdffe` defines a symbol `__xstack` equal to the absolute value `dffe`. The symbol `__xstack` is used by the code generator to initialize the stack pointer.

After you have modified the linker command file, you can link by typing:

```
clnk -o xtest.x12 xtest.lkf
```

For more information about the linker, see **Chapter 6**, “[Using The Linker](#)” and **Chapter 6**, “[Using The Linker](#)” of the “*S12X Compiler User's Guide*”.

Automatic Code and Data Initialization

The XGATE code is designed to be executed from the S12X RAM space. All code and initialized data from the XGATE part of the complete S12X-XGATE application is performed by the S12X startup file. Even if the S12X part has no initialized data, the application must be linked with the **crtsx.x12** startup file, or any user provided startup derived from the *crtsx.s* template file.

For more information, see “*Initializing data in RAM*” in **Chapter 3** and “*Automatic Data Initialization*” in **Chapter 6** of the S12X “*Compiler User’s Guide*”.

Specifying Command Line Options

You specify command line options to **cxxgate** to control the compilation process.

To compile and produce a relocatable file named *xtest.o*, type:

```
cxxgate xtest.c
```

The **-v** option instructs the compiler driver to echo the name and options of each program it calls. The **-l** option instructs the compiler driver to create a mixed listing of C code and assembly language code in the file *xtest.ls*.

To perform the operations described above, enter the command:

```
cxxgate -v -l xtest.c
```

When the compiler exits, the following files are left in your current directory:

- the C source file **xtest.c**
- the C and assembly language listing **xtest.ls**
- the object module **xtest.o**

It is possible to locate listings and object files in specified directories if they are different from the current one, by using respectively the **-cl** and **-co** options:

```
cxxgate -cl\mylist -co\myobj -l xtest.c
```

This command will compile the *xtest.c* file, create a listing named *xtest.ls* in the *\mylist* directory and an object file named *xtest.o* in the *\myobj* directory.

cxxgate allows you to compile more than one file. The input files can be C source files or assembly source files. You can also mix all of these files.

If your application is composed with the following files: two C source files and one assembly source file, you would type:

```
cxxgate -v start.s xtest.c getchar.c
```

This command will assemble the *start.s* file, and compile the two C source files.

See **Chapter 4**, “[Using The Compiler](#)” for information on these and other command line options.

CHAPTER 3

Programming Environments

This chapter explains how to use the COSMIC program development system to perform special tasks required by various S12X-XGATE applications.

Introduction

The XGATE COSMIC compiler is an ANSI C compiler that offers several extensions which support special requirements of embedded systems programmers. This chapter provides details about:

- The const and volatile Type Qualifiers
- Performing Input/Output in C
- Placing Data Objects in The Bss Section
- Redefining Sections
- Referencing Absolute Addresses
- Accessing Internal Registers
- Inserting Inline Assembly Instructions
- Writing Interrupt Handlers
- Placing Addresses in Gate Vectors
- Inline Function
- Interfacing C to Assembly Language
- Register Usage
- Data Representation

The const and volatile Type Qualifiers

You can add the type qualifiers **const** and **volatile** to any base type or pointer type attribute.

Volatile types are useful for declaring data objects that appear to be in conventional storage but are actually represented in machine registers with special properties. You use the type qualifier *volatile* to declare memory mapped input/output control registers, shared data objects, and data objects accessed by signal handlers. The compiler will not optimize references to *volatile* data.

An expression that stores a value in a data object of *volatile* type stores the value immediately. An expression that accesses a value in a data object of *volatile* type obtains the stored value for each access. Your program will not reuse the value accessed earlier from a data object of *volatile* type.

NOTE

*The **volatile** keyword must be used for any data object (variables) that can be modified outside of the normal flow of the function. Without the volatile keyword, all data objects are subject to normal redundant code removal optimizations. Volatile MUST be used for the following conditions:*

- 1) All data objects or variables associated with a memory mapped hardware register e.g. **volatile unsigned int PORTD @0x03;***
- 2) All global variable that can be modified (written to) by an interrupt service routine either directly or indirectly. e.g. a global variable used as a counter in an interrupt service routine.*

You use **const** to declare data objects whose stored values you do not intend to alter during execution of your program. You can therefore place data objects of **const** type in ROM or in write protected program segments. The cross compiler generates an error message if it encounters an expression that alters the value stored in a **const** data object.

If you declare a static data object of *const* type at either file level or at block level, you may specify its stored value by writing a data initializer. The compiler determines its stored value from its data initializer before program startup, and the stored value continues to exist unchanged until program termination. If you specify no data initializer, the stored value is zero. If you declare a data object of *const* type at argument level, you tell the compiler that your program will not alter the value stored in that argument in the related function. If you declare a data object of *const* type and dynamic lifetime at block level, you must specify its stored value by writing a data initializer. If you specify no data initializer, the stored value is undefined.

You may specify *const* and *volatile* together, in either order. A *const volatile* data object could be a Read-only status register, or a variable whose value may be set by another program.

Examples of data objects declared with type qualifiers are:

```
char * const x;    /* const pointer to char */
int * volatile;    /* volatile pointer to int */
const float pi = 355.0/113.0; /* pi is never changed */
```

Performing Input/Output in C

You perform input and output in C by using the C library functions *getchar*, *gets*, *printf*, *putchar*, *puts*, *scanf*, *sprintf* and *sscanf*. They are described in chapter 4.

The C source code for these and all other C library functions is included with the distribution, so that you can modify them to meet your specific needs. Note that all input/output performed by C library functions is supported by underlying calls to *getchar* and *putchar*. These two functions provide access to all input/output library functions. The library is built in such a way so that you need only modify *getchar* and *putchar*, the rest of the library is independent of the runtime environment.

Function definitions for *getchar* and *putchar* are:

```
char getchar(void);
char putchar(char c);
```

Placing Data Objects in The Bss Section

The compiler automatically reserves space for uninitialized data object. All such data are placed in the **.xbss** section. All initialized static data are placed in the **.xdata** section. The **.xbss** section is usually located after the **.xdata** section by the linker command file.

The compiler provides a special option, **+nobss**, which forces uninitialized data to be explicitly located in the **.xdata** section. In such a case, these variables are considered as being explicitly initialized to zero.

Redefining Sections

The compiler uses by default predefined sections to output the various components of a C program. The default sections are:

Section	Description
.xtext	executable code
.xconst	text string and constants
.xdata	initialized variables
.xbss	uninitialized variables

It is possible to redirect any of these components to any user defined section by using the following pragma definition:

```
#pragma section <attribute> <qualified_name>
```

where *<attribute>* is either **empty** or the keyword **const**.

and *<qualified_name>* is a section name enclosed as follows:

```
(name) - parenthesis indicating a code section
[name] - square brackets indicating uninitialized data
{name} - curly braces indicating initialized data
```

A section name is a plain C identifier which **does not** begin with a dot character, and which is no longer than **13** characters. The compiler will prefix automatically the section name with a dot character when passing

this information to the assembler. It is possible to switch back to the default sections by omitting the section name in the *<qualified_name>* sequence.

Each `pragma` directive starts redirecting the selected component from the next declarations. Redefining the `xbss` section forces the compiler to produce the memory definitions for all the previous `xbss` declarations before to switch to the new section.

The following directives:

```
#pragma section (xcode)
#pragma section const {xstring}
#pragma section [xudata]
#pragma section {xidata}
```

redefine the default sections (or the previous one) as following:

- executable code is redirected to section **.xcode**
- strings and constants are redirected to section **.xstring**
- uninitialized variables are redirected to section **.xudata**
- initialized data are redirected to section **.xidata**

Note that **{name}** and **[name]** are equivalent for the constant section as it is always considered as initialized.

The following directive:

```
#pragma section ()
```

switches back the code section to the default section **.xtext**.

Referencing Absolute Addresses

This C compiler allows you to read from and write to absolute addresses, and to assign an absolute address to a function entry point or to a data object. You can give a memory location a symbolic name and associated type, and use it as you would do with any C identifier. This feature is useful for accessing memory mapped I/O ports or for calling functions at known addresses in ROM.

References to absolute addresses have the general form **@<address>**, where <address> is a valid memory location in your environment. For example, to associate an I/O port at address **0x0** with the identifier name *PTA*, write a definition of the form:

```
char PTA @0x0;
```

where **@0x0** indicates an absolute address specification and not a data initializer. Since input/output on the XGATE architecture is memory mapped, performing I/O in this way is equivalent to writing in any given location in memory.

Such a declaration does not reserve any space in memory. The compiler still creates a label, using an *equate* definition, in order to reference the C object symbolically. This symbol is made *public* to allow external usage from any other file.

To use the I/O port in your application, write:

```
char c;
c = PTA; /* to read from input port */
PTA = c; /* to write to output port */
```

Another solution is to use a **#define** directive with a cast to the type of the object being accessed, such as:

```
#define PTA *(char *)0x0
```

which is both inelegant and confusing. The COSMIC implementation is more efficient and easier to use, at the cost of a slight loss in portability. Note that COSMIC C does support the pointer and **#define** methods of implementing I/O access.

Accessing Internal Registers

All registers are declared in the “**iox*.h**” files provided with the compiler. These files should be included in each file using the input-output registers, for example by a:

```
#include <ioxdp512.h>
```

All the register names are defined by assembly equates which are made public. This allows any assembler source to use directly the input-output register names by defining them with an **xref** directive. The XGATE header files are the same as the S12X header files and define all the available registers.

Inserting Inline Assembly Instructions

The compiler features two ways to insert assembly instructions in a C file. The first method uses **#pragma** directives to enclose assembly instructions. The second method uses a special function call to insert assembly instructions. The first one is more convenient for large sequences but does not provide any connection with C object. The second one is more convenient to interface with C objects but is more limited regarding the code length.

Inlining with pragmas

The compiler accepts the following pragma sequences to start and finish assembly instruction blocks:

Directive	Description
#pragma asm	start assembler block
#pragma endasm	end assembler block

The compiler also accepts shorter sequences with the same meaning:

Directive	Description
#asm	start assembler block
#endasm	end assembler block

Such an assembler block may be located anywhere, inside or outside a function. Outside a function, it behaves syntactically as a declaration. This means that such an assembler block cannot split a C declaration somewhere in the middle. Inside a function, it behaves syntactically as one C instruction. This means that there is no trailing semicolon at the end, and no need for enclosing braces. It also means that such an assembler block **cannot** split a C instruction or expression somewhere in the middle.

The following example shows a correct syntax:

```
#pragma asm
    xref asmvar
#pragma endasm

extern char test;

void func(void)
{
    if (test)
    #asm                /* no need for { */
        ldw r2,#asmvar ; access assembler variable
        tfr r3,ccr     ; get flags
        stb r3,(r2)    ; store flags
    #endasm
    else
        test = 1;
}
```

NOTE

*Preprocessing directives are still handled inside assembly code, but `#define` symbols or macros are not replaced within assembly instruction and operands by default. In order to enable such a replacement in the assembly code, the compiler must be run with the **-pad** option. This expansion is limited to the simple macros (without arguments).*

Inlining with `_asm`

The `_asm()` function inserts inline assembly code in your C program. The syntax is:

```
_asm("string constant", arguments...);
```

The “string constant” argument is the assembly code you want embedded in your C program. “arguments” follow the standard C rules for passing arguments. The string you specify follows standard C rules.

NOTE

*The argument string **must** be shorter than **255** characters. If you wish to insert longer assembly code strings you will have to split your input among consecutive calls to `_asm()`.*

For example, carriage returns can be denoted by the ‘\n’ character. For example, to produce the following assembly sequence:

```
ldw r6, #_main
jal r6
```

you would write:

```
_asm("ldw r6, #_main\njal r6\n");
```

The ‘\n’ character is used to separate the instructions when writing multiple instructions in the same line.

To transfer a copy of the condition codes from a C variable to the **ccr** register, you would write:

```
_asm("tfr ccr, r2\n", varcc);
```

`_asm()` does not perform any checks on its argument string. Only the assembler can detect errors in code passed as argument to an `_asm()` call.

`_asm()` can be used in expressions, if the code produced by `_asm` complies with the rules for function returns. For example:

```
if (_asm("tfr r2, ccr\n") & 0x04)
```

allows to test the overflow bit. That way, you can use `_asm()` to write equivalents of C functions directly in assembly language.

NOTE

*With both methods, the assembler source is added as is to the code during the compilation. The optimizer **does not** modify the specified instructions, unless the **-a** option is specified on the code generator. The assembler input can use lowercase or uppercase mnemonics, and may include assembler comments.*

By default, `_asm()` is returning an `int` as any undeclared function. To avoid the need of several definitions (usually conflictuous) when `_asm()` is used with different return types, the compiler implements a special behaviour when a cast is applied to `_asm()`. In such a case, the cast is considered to define the return type of `_asm()` instead of asking for a type conversion. There is no need for any prototype for the `_asm()` function as the parser verifies that the first argument is a string constant.

Inlining Labels

When labels are necessary in the inlined assembly code, the compiler provides a special syntax allowing local labels to be created and handled without interaction with other labels and the optimizer. The sequence `$N` in the assembly source is replaced by a new label name while the sequence `$L` is replaced by the label name created by the last `$N`. Using this syntax, a simple wait loop may be entered as follow:

```
#asm
    ldw   r2,#7
$N:
    sub   r2,#1
    bne   $L    ; loop on the previous label
#endasm
```

Writing Interrupt Handlers

A function declared with the type qualifier **@interrupt** is suitable for direct connection to an XGATE interrupt. **@interrupt** functions cannot return any value but may receive argument through the gate mechanism initializing the **r1** register to point at a prepared structure in the XGATE or the S12X space. The XGATE compiler accepts either a structure name or a list of parameters matching the structure fields as arguments. If no argument is specified, the **r1** register becomes available for code generation.

When you define an **@interrupt** function, the compiler uses the “**rts**” instruction for the return sequence and if necessary, initializes the **r7** stack pointer with the **__xstack** symbol value. This symbol is usually defined in the linker command file. The name of that predefined symbol can be changed by using the **-st** option of the code generator.

You define an **@interrupt** function by using the type qualifier **@interrupt** to qualify the type returned by the function you declare. An example of such a definition is:

```
@interrupt void it_handler(void)
{
    ...
}
```

NOTE

The **@interrupt** function is an extension to the ANSI standard.

Placing Addresses in Gate Vectors

You may use either an assembly language program or a C program to place the addresses of interrupt handlers in interrupt vectors. The assembly language program would be similar to the following example:

```
switch .xconst
xref handler1, handler2, handler3
xref param1, param2, param3
gate1:dc.w handler1, param1
gate2:dc.w handler2, param2
gate3:dc.w handler3, param3
end
```

where *handler1* and so forth are interrupt handlers.

C code that performs the same operation is:

```
struct gate {
    void (*xpc)();
    void *xrl;
};

extern void handler1(), handler2(), handler3();
extern int param1, param2, param3;
const struct xgate xvector[] =
{
    {handler1, &param1},
    {handler2, &param2},
    {handler3, &param3}
};
```

where *handler1* and so forth are interrupt handlers. Then, at link time, this object will be part of the *.xconst* segment, and its address will be used to initialize the XGATE vectors register. A complete gate file example is provided with the compiler (*xvector.c*).

Inline Function

The compiler is able to inline a function body instead of producing a function call. This feature allows the program to run faster but produces a larger code. A function to be inlined has to be defined with the **@inline** modifier. Such a function is kept by the compiler and does not produce any code yet. Each time this function is called in the same source file, the call is replaced by the full body of the inlined function. Because inlined functions are in fact local to a source file, they should be defined in a header file if they have to be used by several source files. To allow the arguments to be passed properly, inlined functions must be defined with prototypes.

NOTE

Inline functions cannot declare static local variables and cannot call themselves either directly or indirectly.

The compiler allows access to specific instructions or features of the XGATE processor, using **@inline** functions. Such functions shall be declared as external functions with the **@inline** modifier. The compiler recognizes the following inline functions:

```
@inline void _sif();
@inline int _par(int);
@inline int _carry(void);
@inline int _bfft(int);
@inline void _csem(int);
@inline int _ssem(int);
```

_sif

the **_sif** function is used to produce the SIF instruction. When used without argument, a single SIF instruction is produced. When used with one argument, it is loaded into a register and the register SIF is produced. The **_sif** function does not return any value or condition.

_par

the **_par** function is used to test or get the parity of its argument expression using the PAR instruction. If the **_par** function is used in a test, the compiler produces a **bcc** or **bcs** instruction. If the **_par** function is used in any other expression, the compiler produces a code sequence setting a register to 0 or 1 depending on the carry bit value.

- `_carry`** the `_carry` function is used to test or get the carry bit from the condition register. If the `_carry` function is used in a test, the compiler produces a **`bcc`** or **`bcs`** instruction. If the `_carry` function is used in any other expression, the compiler produces a code sequence setting a register to 0 or 1 depending on the carry bit value.

- `_bfft`** the `_bfft` function is used to produce the BFFT instruction. The argument expression is loaded into a register and the `_bfft` function returns the result of the BFFT instruction in a register (the same or a different one depending on the following code).

- `_csem`** the `_csem` function is used to clear a semaphore whose number is provided in argument. The `_csem` function does not return any value or condition.

- `_ssem`** the `_ssem` function is used set a semaphore whose number is provided in argument and also sets the carry flag. If the `_ssem` function is used in a test, the compiler produces a **`bcc`** or **`bcs`** instruction. If the `_ssem` function is used in any other expression, the compiler produces a code sequence setting a register to 0 or 1 depending on the carry bit value.

These functions are predeclared in the `processor.h` header file. A full description with examples is provided in Chapter 4.

Interfacing C to Assembly Language

The C cross compiler translates C programs into assembly language according to the specifications described in this section.

You may write external identifiers in both uppercase and lowercase. The compiler prepends an underscore ‘_’ character to each identifier.

The compiler places function code in the **.xtext** section. Function code is not to be altered or read as data. External function names are published via **xdef** declarations.

Literal data such as strings, float or long constants, and switch tables, are normally generated into the **.xconst** section. An option on the code generator allows such constants to be produced in the **.xtext** section.

The compiler generates initialized data into the **.xdata** section. Such external data names are published via **xref** declarations. Data you declare to be of “const” type by adding the type qualifier *const* to its base type is normally generated into the **.xconst** section. Uninitialized data are normally generated into the **.xbss** section unless forced to the **.xdata** section by the compiler option **+nobss**.

Section	Declaration	Reference
.xdata	int init = 1	xdef
.xbss	int uninit	xdef
.xtext	char putchar(c);	xdef
.xconst	const int cx = 2;	xdef
Any of above	extern int out;	xref

Function calls are performed according to the following steps:

- 1) Arguments are moved onto the stack from right to left. Unless the function returns a double or a structure, the first argument is stored in the **r2** register if its size is less than or equal to the size of an *int*, or in **r2,r3** register pair if its type is long or float.

NOTE

There is no widening as expected by the standard ANSI requirements because @interrupt functions are receiving arguments from data structures. The arguments display on the stack is then matching the structure fields display in memory. Anyhow, the XGATE hardware needs an even alignment for integers, so both structures and argument frames will support the same alignment whenever necessary.

- 2) A data space address is moved onto the stack if a structure return area is required.
- 3) The function is called using the following instructions:

```
ldw   r6, #_func
jal   r6
```

- 4) The arguments to the function are popped off the stack.

Register Usage

Except for the return value, the registers **r2** to **r6** and the condition codes are undefined on return from a function call. The return value is in **r2** if it is of type char, char widened to short, short, integer or pointer to... , or in the register pair **r2:r3** if it is of type long or float (**r2** is the most significant word). If the function has argument, **r1** is pointing at the first argument and is saved, adjusted and restored by the function call mechanism. Otherwise, **r1** is undefined.

Stack Display

On a normal function entry, if a stack is necessary, it is implemented using the **r7** register, and automatics are allocated by the sequence:

```
sub r7, <#>
```

which reserves <#> bytes onto the stack.

The stack pointer is set to the beginning of the area reserved for automatic data. The assembler symbol **OFST** is set to the size of the space

needed for automatics. Auto storage is on the stack at **OFST-1,sp** and down. If no automatics are used, the stack frame is **not** built.

NOTE

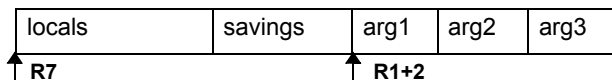
During the function life, the stack pointer may change, so the bias of the same variable may also change to compensate for the difference.

To return, the sequence:

```
add r7,<#>
ldw r6,(r7+)      ; if r6 modified, it was saved
jal r6
```

will restore the previous context. Functions that do not have any arguments or autos, and do not use any temporary storage (required to perform operations on structure data or cast float data, for example) do not create a stack frame and exit simply with a **jal r6**.

The diagrams below show the stack layout at function entry *func*. In this example, *func* has three arguments: *arg1*, *arg2* and *arg3*.



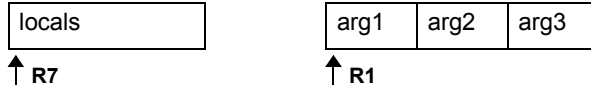
@interrupt functions are in fact entry points of XGATE threads. If they need a stack, they set the stack pointer and reserve a space directly using the sequence:

```
ldw r7,__stack-<#>
```

In any case, because XGATE threads are not interruptible, @interrupt functions are simply returning with the sequence:

```
rts
```

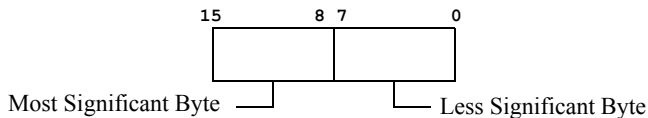
The diagrams below show the stack layout at `@interrupt` function entry *func*, when the `+nofr` option is **NOT** specified. In this example, *func* has three arguments: *arg1*, *arg2* and *arg3*.



When the `+nofr` option is used, an interrupt function can have only one argument which must fit in the **r1** register. In such a case, there is no data area built for that argument which is kept in the **r1** register.

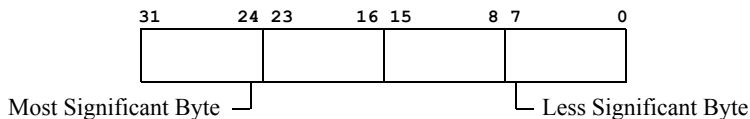
Data Representation

Data objects of type *short int*, *int* and *16 bits pointer* are stored as two bytes, more significant byte first.



Short, Int, 16 bits Pointer

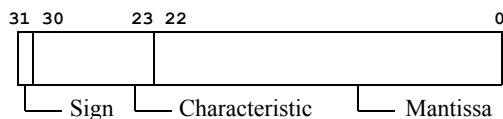
Data objects of type *long integer* and *32 bits pointer* are stored as four bytes, in descending order of significance.



Long, 32 bits Pointer representation

Plain *pointers* are stored as two bytes. *@tiny pointers* (zero page) are stored as one byte.

Data objects of type *float* are represented as for the proposed IEEE Floating Point Standard; four bytes stored in descending order of significance. The IEEE representation is: most significant bit is one for negative numbers, and zero otherwise; the next eight bits are the characteristic, biased such that the binary exponent of the number is the characteristic minus 126; the remaining bits are the fraction, starting with the weighted bit. If the characteristic is zero, the entire number is taken as zero, and should be all zeros to avoid confusing some routines that do not process the entire number. Otherwise there is an assumed 0.5 (assertion of the weighted bit) added to all fractions to put them in the interval [0.5, 1.0). The value of the number is the fraction, multiplied by -1 if the sign bit is set, multiplied by 2 raised to the exponent.



Float representation

CHAPTER 4

Using The Compiler

This chapter explains how to use the C cross compiler to compile programs on your host system. It explains how to invoke the compiler, and describes its options. It also describes the functions which constitute the C library. This chapter includes the following sections:

- Invoking the Compiler
- File Naming Conventions
- Generating Listings
- C Library Support
- Descriptions of C Library Functions

Invoking the Compiler

To invoke the cross compiler, type the command **cxxgate**, followed by the compiler options and the name(s) of the file(s) you want to compile. All the valid compiler options are described in this chapter. Commands to compile source files have the form:

```
cxxgate [options] <files>.[c|s]
```

cxxgate is the name of the *compiler*. The option list is optional. You must include the name of at least one input file *<file>*. *<file>* can be a C source file with the suffix **‘.c’**, or an assembly language source file with the suffix **‘.s’**. You may specify multiple input files with any combination of these suffixes in any order.

If you do not specify any command line options, **cxxgate** will compile your *<files>* with the default options. It will also write the name of each file as it is processed. It writes any error messages to STDERR.

The following command line:

```
cxxgate acia.c
```

compiles and assembles the *acia.c* C program, creating the relocatable program **acia.o**.

If the compiler finds an error in your program, it halts compilation. When an error occurs, the compiler sends an error message to your terminal screen unless the option **-e** has been specified on the command line. In this case, all error messages are written to a file whose name is obtained by replacing the suffix *.c* of the source file by the suffix *.err*. An error message is still output on the terminal screen to indicate that errors have been found. **Appendix A**, “[Compiler Error Messages](#)”, lists the error messages the compiler generates. If one or more command line arguments are invalid, **cxxgate** processes the next file name on the command line and begins the compilation process again.

The example command above does not specify any compiler options. In this case, the compiler will use only default options to compile and

assemble your program. You can change the operation of the compiler by specifying the options you want when you run the compiler.

To specify options to the compiler, type the appropriate option or options on the command line as shown in the first example above. Options should be separated with spaces. You must include the '-' or '+' that is part of the option name.

Compiler Command Line Options

The **cxxgate** compiler accepts the following command line options, each of which is described in detail below:

```

cxxgate [options] <files>
  -a*>  assembler options
  -ce*  path for errors
  -cl*  path for listings
  -co*  path for objects
  -d*>  define symbol
  -e    create error file
  -ec   all C files
  -es   all assembler files
  -ex*  prefix executables
  -f*   configuration file
  -g*>  code generator options
  -i*>  path for include
  -l    create listing
  -no   do not use optimizer
  -o*>  optimizer options
  -p*>  parser options
  -s    create only assembler file
  -sp   create only preprocessor file
  -t*   path for temporary files
  -v    verbose
  -x    do not execute
  +*>   select compiler options

```

-a*> specify assembler options. Up to 60 options can be specified on the same command line. See **Chapter 5**, “[Using The Assembler](#)”, to get the list of all accepted options.

- ce*** specify a path for the error files. By default, errors are created in the same directory than the source files.
- cl*** specify a path for the listing files. By default, listings are created in the same directory than the source files.
- co*** specify a path for the object files. By default, objects are created in the same directory than the source files.
- d*^** specify * as the name of a user-defined preprocessor symbol (**#define**). The form of the definition is **-dsymbol[=value]**; the symbol is set to **1** if value is omitted. You can specify up to 60 such definitions.
- e** log errors from parser in a file instead of displaying them on the terminal screen. The error file name is defaulted to `<file>.err`, and is created only if there are errors.
- ec** treat all files as C source files.
- es** treat all files as assembler source files.
- ex** use the compiler driver's path as prefix to quickly locate the executable passes. Default is to use the path variable environment. This method is faster than the default behaviour but reduces the command line length.
- f*** specify * as the name of a configuration file. This file contains a list of options which will be automatically used by the compiler. If no file name is specified, then the compiler looks for a default configuration file named *cxsgate.cxf* in the compiler directory as specified in the installation process. For more information, see **Appendix B**, "[Modifying Compiler Operation](#)".
- g*>** specify code generation options. Up to 60 options can be specified. See **Appendix D**, "[Compiler Passes](#)", for the list of all accepted options.

- i*>** define include path. You can define up to 60 different paths. Each path is a directory name, **not** terminated by any directory separator character.
- l** merge C source listing with assembly language code; listing output defaults to `<file>.ls`.
- no** do not use the optimizer.
- o*>** specify optimizer options. Up to 60 options can be specified. See **Appendix D**, “[Compiler Passes](#)”, for the list of all accepted options.
- p*>** specify parser options. Up to 60 options can be specified. See **Appendix D**, “[Compiler Passes](#)”, for the list of all accepted options.
- s** create only assembler files and stop. Do not assemble the files produced.
- sp** create only preprocessed files and stop. Do not compile files produced. Preprocessed output defaults to `<file>.p`. The produced files can be compiled as C source files.
- t*** specify path for temporary files. The path is a directory name, **not** terminated by any directory separator character.
- v** be “verbose”. Before executing a command, print the command, along with its arguments, to STDOUT. The default is to output only the names of each file processed. Each name is followed by a colon and newline.
- x** do not execute the passes, instead, write to STDOUT the commands which otherwise would have been performed.
- +*>** select a predefined compiler option. These options are predefined in the configuration file. You can specify up to 20 compiler options on the command line. The following documents the available options as provided by the default configuration file:

- +debug** produce debug information to be used by the debug utilities provided with the compiler and by any external debugger.
- +nobss** do not use the *.bss* section. By default, uninitialized variables are defined into the *.bss* section. This option is useful to force all variables to be grouped into a single section.
- +nocst** output literals and constants in the code section *.text* instead of the specific section *.const*.
- +nofr** uses *r1* as the only argument of interrupt functions. By default, *r1* is handled as a frame pointer and allows several arguments to be passed to interrupt functions.
- +proto** enforce prototype declaration for functions. An error message is issued if a function is used and no prototype declaration is found for it. By default, the compiler accepts both syntaxes without any error.
- +rev** reverse the bitfield filling order. By default, bitfields are filled from the less significant bit (LSB) towards the most significant bit (MSB) of a memory cell. If the *+rev* option is specified, bitfields are filled from the *msb* to the *lsb*.
- +v1** produce code for first silicon versions. By default, code is targeting second silicon versions.
- +warn** enable warnings.

File Naming Conventions

The programs making up the C cross compiler generate the following output file names, by default. See the documentation on a specific program for information about how to change the default file names accepted as input or generated as output.

Program	Input File Name	Output File Name
cpxgate	<file>.c	<file>.1
cgxgate	<file>.1	<file>.2
coxgate	<file>.2	<file>.s
error listing	<file>.c	<file>.err
assembler listing	<file>.[c s]	<file>.ls
C header files	<file>.h	

caxgate	<file>.s	<file>.o
source listing	<file>.s	<file>.ls

clnk	<file>.o	name required
-------------	----------	---------------

Generating Listings

You can generate listings of the output of any (or all) the compiler passes by specifying the **-l** option to **cxxgate**. You can locate the listing file in a different directory by using the **-cl** option.

The example program provided in the package shows the listing produced by compiling the C source file *acia.c* with the **-l** option:

```
cxxgate -l acia.c
```

Generating an Error File

You can generate a file containing all the error messages output by the parser by specifying the **-e** option to **cxxgate**. You can locate the error file in a different directory by using the **-ce** option. For example, you would type:

```
cxxgate -e prog.c
```

The error file name is obtained from the source filename by replacing the *.c* suffix by the *.err* suffix.

Return Status

cxxgate returns success if it can process all files successfully. It prints a message to `STDERR` and returns failure if there are errors in at least one processed file.

Examples

To echo the names of each program that the compiler runs:

```
cxxgate -v acia.c
```

To save the intermediate files created by the code generator and halt before the assembler:

```
cxxgate -s file.c
```

C Library Support

This section describes the facilities provided by the C library. The C cross compiler for XGATE includes all useful functions for programmers writing applications for ROM-based systems.

How C Library Functions are Packaged

The functions in the C library are packaged in three separate sub-libraries, one for machine-dependent routines (the [machine](#) library), one that does not support floating point (the [integer](#) library) and one that provides full floating point support (the [floating point](#) library). If your application does not perform floating point calculations, you can decrease its size and increase its runtime efficiency by including only the integer library.

Inserting Assembler Code Directly

Assembler instructions can be quoted directly into C source files, and entered unchanged into the output assembly stream, by use of the [_asm\(\)](#) function. This function is not part of any library as it is recognized by the compiler itself. See “[Inserting Inline Assembly Instructions](#)” in [Chapter 3](#)

Linking Libraries with Your Program

If your application requires floating point support, you must specify the floating point library [before](#) the integer library in the linker command file. Modules common to both libraries will therefore be loaded from the floating point library, followed by the appropriate modules from the floating point and integer libraries, in that order.

Integer Library Functions

The following table lists the C library functions in the integer library.

_asm	isprint	putchar	strcspn
abs	ispunct	puts	strlen
atoi	isspace	rand	strncat
atol	isupper	realloc	strncmp
calloc	isxdigit	sbreak	strncpy
free	labs	scanf	strpbrk
getchar	longjmp	setjmp	strchr
gets	malloc	sprintf	strspn
isalnum	memchr	srand	strstr

<code>isalpha</code>	<code>memcmp</code>	<code>sscanf</code>	<code>strtol</code>
<code>iscntrl</code>	<code>memcpy</code>	<code>strcat</code>	<code>tolower</code>
<code>isdigit</code>	<code>memmove</code>	<code>strchr</code>	<code>toupper</code>
<code>isgraph</code>	<code>memset</code>	<code>strcmp</code>	<code>vprintf</code>
<code>islower</code>	<code>printf</code>	<code>strcpy</code>	<code>vsprintf</code>

Floating Point Library Functions

The following table lists the C library functions in the float library.

<code>acos</code>	<code>exp</code>	<code>modf</code>	<code>sscanf</code>
<code>asin</code>	<code>fabs</code>	<code>pow</code>	<code>strtod</code>
<code>atan</code>	<code>floor</code>	<code>printf</code>	<code>tan</code>
<code>atan2</code>	<code>fmod</code>	<code>scanf</code>	<code>tanh</code>
<code>atof</code>	<code>frexp</code>	<code>sin</code>	<code>vprintf</code>
<code>ceil</code>	<code>ldexp</code>	<code>sinh</code>	<code>vsprintf</code>
<code>cos</code>	<code>log</code>	<code>sprintf</code>	
<code>cosh</code>	<code>log10</code>	<code>sqrt</code>	

Common Input/Output Functions

Six of the functions that perform stream input/output are included in both the integer and floating point libraries. The functionalities of the versions in the integer library are a subset of the functionalities of their floating point counterparts. The versions in the integer library cannot print or manipulate floating point numbers. These functions are: *printf*, *scanf*, *sprintf*, *sscanf*, *vprintf* and *vsprintf*.

Functions Implemented as Macros

Five of the functions in the C library are actually implemented as “macros”. Unlike other functions, which (if they do not return *int*) are declared in header files and defined in a separate object module that is linked in with your program later, functions implemented as macros are defined using **#define** preprocessor directives in the header file that declares them. Macros can therefore be used independently of any library by including the header file that defines and declares them with your program, as explained below. The functions in the C library that are implemented as macros are: *max*, *min*, *va_arg*, *va_end*, and *va_start*.

Including Header Files

If your application calls a C library function, you must include the header file that declares the function at compile time, in order to use the proper return type and the proper function prototyping, so that all the

expected arguments are properly evaluated. You do this by writing a preprocessor directive of the form:

```
#include <header_name>
```

in your program, where *<header_name>* is the name of the appropriate header file enclosed in angle brackets. The required header file should be included before you refer to any function that it declares.

The names of the header files packaged with the C library and the functions declared in each header are listed below.

<assert.h> - Header file for the assertion macro: *assert*.

<ctype.h> - Header file for the character functions: *isalnum*, *isalpha*, *isctrl*, *isgraph*, *isprint*, *ispunct*, *isspace*, *isxdigit*, *isdigit*, *isupper*, *islower*, *tolower* and *toupper*.

<float.h> - Header file for limit constants for floating point values.

<iox*.h> - Header file for input-output registers. Each register has an upper-case name which matches the standard Motorola definition.

<limits.h> - Header file for limit constants of the compiler.

<math.h> - Header file for mathematical functions: *acos*, *asin*, *atan*, *atan2*, *ceil*, *cos*, *cosh*, *exp*, *fabs*, *floor*, *fmod*, *frexp*, *ldexp*, *log*, *log10*, *modf*, *pow*, *sin*, *sinh*, *sqrt*, *tan* and *tanh*.

<processor.h> - Header file for **inline** functions: *_sif*, *_par*, *_carry*, *_bfoo*, *_csem*, *_ssem*.

<setjmp.h> - Header file for nonlocal jumps: *setjmp* and *longjmp*.

<stdarg.h> - Header file for walking argument lists: *va_arg*, *va_end* and *va_start*. Use these macros with any function you write that must accept a variable number of arguments.

<stddef.h> - Header file for types: *size_t*, *wchar_t* and *ptrdiff_t*.

<stdio.h> - Header file for stream input/output: *getchar*, *gets*, *printf*, *putchar*, *puts*, *scanf*, *sprintf*, *sscanf*, *vprintf* and *vsprintf*.

<stdlib.h> - Header file for general utilities: *abs*, *abort*, *atof*, *atoi*, *atol*, *calloc*, *div*, *exit*, *free*, *isqrt*, *labs*, *ldiv*, *lsqrt*, *malloc*, *rand*, *realloc*, *srand*, *strtod*, *strtol* and *strtoul*.

<string.h> - Header file for string functions: *memchr*, *memcmp*, *memcpy*, *memmove*, *memset*, *strcat*, *strchr*, *strcmp*, *strcpy*, *strcspn*, *strlen*, *strncat*, *strncmp*, *strncpy*, *strpbrk*, *strrchr*, *strspn* and *strstr*.

Functions returning int - C library functions that return *int* and can therefore be called without any header file, since *int* is the function return type that the compiler assumed by default, are: *isalnum*, *isalpha*, *isctrl*, *isgraph*, *isprint*, *ispunct*, *isspace*, *isxdigit*, *isdigit*, *isupper*, *islower*, *sbreak*, *tolower* and *toupper*.

Descriptions of C Library Functions

The following pages describe each of the functions in the C library in quick reference format. The descriptions are in alphabetical order by function name.

The *syntax* field describes the function prototype with the return type and the expected arguments, and if any, the header file name where this function has been declared.

`_asm`

Description

Generate inline assembly code

Syntax

```
_asm("string constant", arguments...)
```

Function

`_asm()` generates inline assembly code by copying `<string constant>` and quoting it into the output assembly code stream. `<arguments>` are first evaluated following the usual rules for passing arguments. The first argument is kept in the `a` register whenever possible, and all other arguments are pushed onto the stack. After the `<string constant>` code is output, arguments pushed to the stack are removed before to continue.

For more information, see “[Inserting Inline Assembly Instructions](#)” in **Chapter 3**.

Return Value

Nothing, unless `_asm()` is used in an expression. In that case, normal return conventions must be followed. See “[Register Usage](#)” in **Chapter 3**.

Example

The sequence `ldw r6,#_main,jal r6`, may be generated by the following call:

```
_asm("ldw r6,#_main\njal r6\n");
```

Note that the string-quoting syntax matches the familiar `printf()` function.

Notes

`_asm()` is not packaged in any library. It is recognized by the compiler itself.

abort

Description

Abort program execution

Syntax

```
#include <stdlib.h>
void abort(void)
```

Function

abort stops the program execution by calling the *exit* function which is placed by the startup module just after the call to the main function.

Return Value

abort never returns.

Example

To abort in case of error:

```
if (fatal_error)
    abort();
```

See Also

exit

Notes

abort is a macro equivalent to the function name *exit*.

abs

Description

Find absolute value

Syntax

```
#include <stdlib.h>
int abs(int i)
```

Function

abs obtains the absolute value of **i**. No check is made to see that the result can be properly represented.

Return Value

abs returns the absolute value of **i**, expressed as an int.

Example

To print out a debit or credit balance:

```
printf("balance %d%s\n", abs(bal), (bal < 0)? "CR" : "");
```

See Also

labs, fabs

Notes

abs is packaged in the integer library.

acos

Description

Arccosine

Syntax

```
#include <math.h>
double acos(double x)
```

Function

acos computes the angle in radians the cosine of which is **x**, to full double precision.

Return Value

acos returns the closest internal representation to *acos(x)*, expressed as a double floating value in the range [0, pi]. If **x** is outside the range [-1, 1], *acos* returns zero.

Example

To find the arccosine of **x**:

```
theta = acos(x);
```

See Also

asin, *atan*, *atan2*

Notes

acos is packaged in the floating point library.

asin

Description

Arcsine

Syntax

```
#include <math.h>
double asin(double x)
```

Function

asin computes the angle in radians the sine of which is **x**, to full double precision.

Return Value

asin returns the nearest internal representation to *asin(x)*, expressed as a double floating value in the range $[-\pi/2, \pi/2]$. If **x** is outside the range $[-1, 1]$, *asin* returns zero.

Example

To compute the arcsine of **y**:

```
theta = asin(y);
```

See Also

acos, *atan*, *atan2*

Notes

asin is packaged in the floating point library.

atan

Description

Arctangent

Syntax

```
#include <math.h>
double atan(double x)
```

Function

atan computes the angle in radians; the tangent of which is **x**, *atan* computes the angle in radians; the tangent of which is **x**, to full double precision.

Return Value

atan returns the nearest internal representation to *atan(x)*, expressed as a double floating value in the range $[-\pi/2, \pi/2]$.

Example

To find the phase angle of a vector in degrees:

```
theta = atan(y/x) * 180.0 / pi;
```

See Also

acos, *asin*, *atan2*

Notes

atan is packaged in the floating point library.

atan2

Description

Arctangent of y/x

Syntax

```
#include <math.h>
double atan2(double y, double x)
```

Function

atan2 computes the angle in radians the tangent of which is y/x to full double precision. If y is negative, the result is negative. If x is negative, the magnitude of the result is greater than $\pi/2$.

Return Value

atan2 returns the closest internal representation to $\text{atan}(y/x)$, expressed as a double floating value in the range $[-\pi, \pi]$. If both input arguments are zero, *atan2* returns zero.

Example

To find the phase angle of a vector in degrees:

```
theta = atan2(y/x) * 180.0/pi;
```

See Also

acos, *asin*, *atan*

Notes

atan2 is packaged in the floating point library.

atof

Description

Convert buffer to double

Syntax

```
#include <stdlib.h>
double atof(char *nptr)
```

Function

atof converts the string at *nptr* into a double. The string is taken as the text representation of a decimal number, with an optional fraction and exponent. Leading whitespace is skipped and an optional sign is permitted; conversion stops on the first unrecognizable character. Acceptable inputs match the pattern:

$$[+|-]d*.[d*][e[+|-]dd*]$$

where **d** is any decimal digit and **e** is the character ‘e’ or ‘E’. No checks are made against overflow, underflow, or invalid character strings.

Return Value

atof returns the converted double value. If the string has no recognizable characters, it returns zero.

Example

To read a string from STDIN and convert it to a double at **d**:

```
gets(buf);
d = atof(buf);
```

See Also

atoi, atol, strtol, strtod

Notes

atof is packaged in the floating point library.

atoi

Description

Convert buffer to integer

Syntax

```
#include <stdlib.h>
int atoi(char *nptr)
```

Function

atoi converts the string at *nptr* into an integer. The string is taken as the text representation of a decimal number. Leading whitespace is skipped and an optional sign is permitted; conversion stops on the first unrecognizable character. Acceptable characters are the decimal digits. If the stop character is **I** or **L**, it is skipped over.

No checks are made against overflow or invalid character strings.

Return Value

atoi returns the converted integer value. If the string has no recognizable characters, zero is returned.

Example

To read a string from STDIN and convert it to an *int* at **i**:

```
gets(buf);
i = atoi(buf);
```

See Also

atof, *atol*, *strtol*, *strtod*

Notes

atoi is packaged in the integer library.

atol

Description

Convert buffer to long

Syntax

```
#include <stdlib.h>
long atol(char *nptr)
```

Function

atol converts the string at *nptr* into a long integer. The string is taken as the text representation of a decimal number. Leading whitespace is skipped and an optional sign is permitted; conversion stops on the first unrecognizable character. Acceptable characters are the decimal digits. If the stop character is **I** or **L** it is skipped over.

No checks are made against overflow or invalid character strings.

Return Value

atol returns the converted long integer. If the string has no recognizable characters, zero is returned.

Example

To read a string from STDIN and convert it to a long **I**:

```
gets(buf);
l = atol(buf);
```

See Also

atof, *atoi*, *strtol*, *strtod*

Notes

atol is packaged in the integer library.

`_bfft`

Description

Get the first bit set position

Syntax

```
#include <processor.h>
@inline int _bfft(int)
```

Function

`_bfft` is an inline function allowing to get the position of the first bit set in its argument expression, by using a BFFT instruction. The carry flag is also set but cannot be tested directly as for the other flags (zero, negative). If necessary, it can be tested by the `_carry` inline function.

Return Value

`_bfft` returns the BFFT instruction result in a register and sets the flags accordingly.

Example

<code>n = _bfft(v);</code>	produces	<code>ldw r2, (r1, #2)</code>
		<code>bfft r3, r2</code>
		<code>stw r3, (r1, #4)</code>

See Also

`_carry`

Notes

`_bfft` is an inline function and then is not defined in any library. It is therefore not possible to take its address. For more information, see “[Inline Function](#)” in **Chapter 3**.

calloc

Description

Allocate and clear space on the heap

Syntax

```
#include <stdlib.h>
void *calloc(int nelem, int elsize)
```

Function

calloc allocates space on the heap for an item of size *nbytes*, where *nbytes* = *nelem* * *elsize*. The space allocated is guaranteed to be at least *nbytes* long, starting from the pointer returned, which is guaranteed to be on a proper storage boundary for an object of any type. The heap is grown as necessary. If space is exhausted, *calloc* returns a null pointer. The pointer returned may be assigned to an object of any type without casting. The allocated space is initialized to zero.

Return Value

calloc returns a pointer to the start of the allocated cell if successful; otherwise it returns NULL.

Example

To allocate an array of ten doubles:

```
double *pd;
pd = calloc(10, sizeof (double));
```

See Also

free, malloc, realloc

Notes

calloc is packaged in the integer library.

`_carry`

Description

Test or get the carry bit

Syntax

```
#include <processor.h>
@inline int _carry(void)
```

Function

`_carry` is an inline function allowing to test or get the value of the *carry* bit. When used in an *if* construct, this function expands directly to a **bcc** or **bcs** instruction. When used in an expression, it expands in order to build in a register the value **0** or **1** depending on the *carry* bit value.

Return Value

`_carry` returns **0** or **1** in the a register if such a value is needed.

Example

low <= 1;	produces	lsl r2, #1
if (_carry())		bcc L1
++high;		add r3, #1
	L1:	
low <= 1;	produces	lsl r2, #1
high = _carry()		adc r3, r0, r0

See Also

`_bfft`

Notes

`_carry` is an inline function and then is not defined in any library. It is therefore not possible to take its address. For more information, see “[Inline Function](#)” in **Chapter 3**.

ceil

Description

Round to next higher integer

Syntax

```
#include <math.h>
double ceil(double x)
```

Function

ceil computes the smallest integer greater than or equal to **x**.

Return Value

ceil returns the smallest integer greater than or equal to **x**, expressed as a double floating value.

Example

x	ceil(x)
5.1	6.0
5.0	5.0
0.0	0.0
-5.0	-5.0
-5.1	-5.0

See Also

floor

Notes

ceil is packaged in the floating point library.

COS

Description

Cosine

Syntax

```
#include <math.h>
double cos(double x)
```

Function

cos computes the cosine of **x**, expressed in radians, to full double precision. If the magnitude of **x** is too large to contain a fractional quadrant part, the value of *cos* is 1.

Return Value

cos returns the nearest internal representation to $\cos(x)$ in the range $[0, \pi]$, expressed as a double floating value. A large argument may return a meaningless value.

Example

To rotate a vector through the angle **theta**:

```
xnew = xold * cos(theta) - yold * sin(theta);
ynew = xold * sin(theta) + yold * cos(theta);
```

See Also

sin, *tan*

Notes

cos is packaged in the floating point library.

cosh

Description

Hyperbolic cosine

Syntax

```
#include <math.h>
double cosh(double x)
```

Function

cosh computes the hyperbolic cosine of **x** to full double precision.

Return Value

cosh returns the nearest internal representation to *cosh(x)* expressed as a double floating value. If the result is too large to be properly represented, *cosh* returns zero.

Example

To use the Moivre's theorem to compute $(\cosh x + \sinh x)$ to the *n*th power:

```
demoivre = cosh(n * x) + sinh(n * x);
```

See Also

exp, *sinh*, *tanh*

Notes

cosh is packaged in the floating point library.

`_csem`

Description

Clear a semaphore

Syntax

```
#include <processor.h>
@inline void _csem(int)
```

Function

`_csem` is an inline function allowing to clear a semaphore given by the argument, by using the CSEM instruction.

Return Value

`_csem` returns nothing.

Example

<code>_csem(3);</code>	produces	<code>csem #3</code>
<code>_csem(s);</code>	produces	<code>ldw r2, (r1, #2)</code> <code>csem r2</code>

Notes

`_csem` is an inline function and then is not defined in any library. It is therefore not possible to take its address. For more information, see “[Inline Function](#)” in **Chapter 3**.

div

Description

Divide with quotient and remainder

Syntax

```
#include <stdlib.h>
div_t div(int numer, int denom)
```

Function

div divides the integer *numer* by the integer *denom* and returns the quotient and the remainder in a structure of type *div_t*. The field *quot* contains the quotient and the field *rem* contains the remainder.

Return Value

div returns a structure of type *div_t* containing both quotient and remainder.

Example

To get minutes and seconds from a delay in seconds:

```
div_t result;

result = div(time, 60);
min = result.quot;
sec = result.rem;
```

See Also

ldiv

Notes

div is packaged in the integer library.

exit

Description

Exit program execution

Syntax

```
#include <stdlib.h>
void exit(int status)
```

Function

exit stops the execution of a program by switching to the startup module just after the call to the *main* function. The status argument is not used by the current implementation.

Return Value

exit never returns.

Example

To *exit* in case of error:

```
if (fatal_error)
    exit();
```

See Also

abort

Notes

exit is in the startup module.

exp

Description

Exponential

Syntax

```
#include <math.h>
double exp(double x)
```

Function

exp computes the exponential of **x** to full double precision.

Return Value

exp returns the nearest internal representation to *exp* **x**, expressed as a double floating value. If the result is too large to be properly represented, *exp* returns zero.

Example

To compute the hyperbolic sine of **x**:

```
sinh = (exp(x) - exp(-x)) / 2.0;
```

See Also

log

Notes

exp is packaged in the floating point library.

fabs

Description

Find double absolute value

Syntax

```
#include <math.h>
double fabs(double x)
```

Function

fabs obtains the absolute value of **x**.

Return Value

fabs returns the absolute value of **x**, expressed as a double floating value.

Example

x	fabs(x)
5.0	5.0
0.0	0.0
-3.7	3.7

See Also

abs, *labs*

Notes

fabs is packaged in the floating point library.

floor

Description

Round to next lower integer

Syntax

```
#include <math.h>
double floor(double x)
```

Function

floor computes the largest integer less than or equal to **x**.

Return Value

floor returns the largest integer less than or equal to **x**, expressed as a double floating value.

Example

x	floor(x)
5.1	5.0
5.0	5.0
0.0	0.0
-5.0	-5.0
-5.1	-6.0

See Also

ceil

Notes

floor is packaged in the floating point library.

fmod

Description

Find double modulus

Syntax

```
#include <math.h>
double fmod(double x, double y)
```

Function

fmod computes the floating point remainder of **x** / **y**, to full double precision. The return value of **f** is determined using the formula:

$$\mathbf{f} = \mathbf{x} - \mathbf{i} * \mathbf{y}$$

where **i** is some integer, **f** is the same sign as **x**, and the absolute value of **f** is less than the absolute value of **y**.

Return Value

fmod returns the value of **f** expressed as a double floating value. If **y** is zero, *fmod* returns zero.

Example

x	y	fmod(x, y)
5.5	5.0	0.5
5.0	5.0	0.0
0.0	0.0	0.0
-5.5	5.0	-0.5

Notes

fmod is packaged in the floating point library.

free

Description

Free space on the heap

Syntax

```
#include <stdlib.h>
void free(void *ptr)
```

Function

free returns an allocated cell to the heap for subsequence reuse. The cell pointer *ptr* must have been obtained by an earlier *calloc*, *malloc*, or *realloc* call; otherwise the heap will become corrupted. *free* does its best to check for invalid values of *ptr*. A NULL value for *ptr* is explicitly allowed, however, and is ignored.

Return Value

Nothing.

Example

To give back an allocated area:

```
free (pd) ;
```

See Also

calloc, *malloc*, *realloc*

Notes

No effort is made to lower the system break when storage is freed, so it is quite possible that earlier activity on the heap may cause problems later on the stack.

free is packaged in the integer library.

frexp

Description

Extract fraction from exponent part

Syntax

```
#include <math.h>
double frexp(double val, int *exp)
```

Function

frexp partitions the double at *val*, which should be non-zero, into a fraction in the interval $[1/2, 1)$ times two raised to an integer power. It then delivers the integer power to **exp*, and returns the fractional portion as the value of the function. The exponent is generally meaningless if *val* is zero.

Return Value

frexp returns the power of two fraction of the double at *val* as the return value of the function, and writes the exponent at **exp*.

Example

To implement the *sqrt(x)* function:

```
double sqrt(double x)
{
    extern double newton(double);
    int n;

    x = frexp(x, &n);
    x = newton(x);
    if (n & 1)
        x *= SQRT2;
    return (ldexp(x, n / 2));
}
```

See Also

ldexp

Notes

frexp is packaged in the floating point library.

getchar

Description

Get character from input stream

Syntax

```
#include <stdio.h>
int getchar(void)
```

Function

getchar obtains the next input character, if any, from the user supplied input stream. This user must rewrite this function in C or in assembly language to provide an interface to the input mechanism of the C library.

Return Value

getchar returns the next character from the input stream. If end of file (break) is encountered, or a read error occurs, *getchar* returns EOF.

Example

To copy characters from the input stream to the output stream:

```
while ((c = getchar()) != EOF)
    putchar(c);
```

See Also

putchar

Notes

getchar is packaged in the integer library, and is by default using the first serial port **SCI 1**.

gets

Description

Get a text line from input stream

Syntax

```
#include <stdio.h>
char *gets(char *s)
```

Function

gets copies characters from the input stream to the buffer starting at **s**. Characters are copied until a newline is reached or end of file is reached. If a newline is reached, it is discarded and a NUL is written immediately following the last character read into **s**.

gets uses *getchar* to read each character.

Return Value

gets returns **s** if successful. If end of file is reached, *gets* returns NULL. If a read error occurs, the array contents are indeterminate and *gets* returns NULL.

Example

To copy input to output, line by line:

```
while (puts(gets(buf)))
    ;
```

See Also

puts

Notes

There is no assured limit on the size of the line read by *gets*.

gets is packaged in the integer library.

isalnum

Description

Test for alphabetic or numeric character

Syntax

```
#include <ctype.h>
int isalnum(int c)
```

Function

isalnum tests whether **c** is an alphabetic character (either upper or lower case), or a decimal digit.

Return Value

isalnum returns nonzero if the argument is an alphabetic or numeric character; otherwise the value returned is zero.

Example

To test for a valid C identifier:

```
if (isalpha(*s) || *s == '_')
    for (++s; isalnum(*s) || *s == '_'; ++s)
        ;
```

See Also

isalpha, *isdigit*, *islower*, *isupper*, *isxdigit*, *tolower*, *toupper*

Notes

If the argument is outside the range [-1, 255], the result is undefined.

isalnum is packaged in the integer library.

isalpha

Description

Test for alphabetic character

Syntax

```
#include <ctype.h>
int isalpha(int c)
```

Function

isalpha tests whether **c** is an alphabetic character, either upper or lower case.

Return Value

isalpha returns nonzero if the argument is an alphabetic character. Otherwise the value returned is zero.

Example

To find the end points of an alphabetic string:

```
while (*first && !isalpha(*first))
    ++first;
for (last = first; isalpha(*last); ++last)
    ;
```

See Also

isalnum, isdigit, islower, isupper, isxdigit, tolower, toupper

Notes

If the argument is outside the range [-1, 255], the result is undefined.

isalpha is packaged in the integer library.

iscntrl

Description

Test for control character

Syntax

```
#include <ctype.h>
int iscntrl(int c)
```

Function

iscntrl tests whether **c** is a delete character (0177 in ASCII), or an ordinary control character (less than 040 in ASCII).

Return Value

iscntrl returns nonzero if **c** is a control character; otherwise the value is zero.

Example

To map control characters to percent signs:

```
for (; *s; ++s)
    if (iscntrl(*s))
        *s = '%';
```

See Also

isgraph, *isprint*, *ispunct*, *isspace*

Notes

If the argument is outside the range [-1, 255], the result is undefined.

iscntrl is packaged in the integer library.

isdigit

Description

Test for digit

Syntax

```
#include <ctype.h>
int isdigit(int c)
```

Function

isdigit tests whether **c** is a decimal digit.

Return Value

isdigit returns nonzero if **c** is a decimal digit; otherwise the value returned is zero.

Example

To convert a decimal digit string to a number:

```
for (sum = 0; isdigit(*s); ++s)
    sum = sum * 10 + *s - '0';
```

See Also

isalnum, *isalpha*, *islower*, *isupper*, *isxdigit*, *tolower*, *toupper*

Notes

If the argument is outside the range $[-1, 255]$, the result is undefined.

isdigit is packaged in the integer library.

isgraph

Description

Test for graphic character

Syntax

```
#include <ctype.h>
int isgraph(int c)
```

Function

isgraph tests whether **c** is a graphic character; i.e. any printing character except a space (040 in ASCII).

Return Value

isgraph returns nonzero if **c** is a graphic character. Otherwise the value returned is zero.

Example

To output only graphic characters:

```
for (; *s; ++s)
    if (isgraph(*s))
        putchar(*s);
```

See Also

iscntrl, *isprint*, *ispunct*, *isspace*

Notes

If the argument is outside the range [-1, 255], the result is undefined.

isgraph is packaged in the integer library.

islower

Description

Test for lowercase character

Syntax

```
#include <ctype.h>
int islower(int c)
```

Function

islower tests whether **c** is a lowercase alphabetic character.

Return Value

islower returns nonzero if **c** is a lowercase character; otherwise the value returned is zero.

Example

To convert to uppercase:

```
if (islower(c))
    c += 'A' - 'a';    /* also see toupper() */
```

See Also

isalnum, *isalpha*, *isdigit*, *isupper*, *isxdigit*, *tolower*, *toupper*

Notes

If the argument is outside the range [-1, 255], the result is undefined.

islower is packaged in the integer library.

isprint

Description

Test for printing character

Syntax

```
#include <ctype.h>
int isprint(int c)
```

Function

isprint tests whether **c** is any printing character. Printing characters are all characters between a space (040 in ASCII) and a tilde '~' character (0176 in ASCII).

Return Value

isprint returns nonzero if **c** is a printing character; otherwise the value returned is zero.

Example

To output only printable characters:

```
for (; *s; ++s)
    if (isprint(*s))
        putchar(*s);
```

See Also

isctrl, *isgraph*, *ispunct*, *isspace*

Notes

If the argument is outside the range [-1, 255], the result is undefined.

isprint is packaged in the integer library.

ispunct

Description

Test for punctuation character

Syntax

```
#include <ctype.h>
int ispunct(int c)
```

Function

ispunct tests whether **c** is a punctuation character. Punctuation characters include any printing character except space, a digit, or a letter.

Return Value

ispunct returns nonzero if **c** is a punctuation character; otherwise the value returned is zero.

Example

To collect all punctuation characters in a string into a buffer:

```
for (i = 0; *s; ++s)
    if (ispunct(*s))
        buf[i++] = *s;
```

See Also

isctrl, *isgraph*, *isprint*, *isspace*

Notes

If the argument is outside the range [-1, 255], the result is undefined.

ispunct is packaged in the integer library.

isqrt

Description

Integer square root

Syntax

```
#include <stdlib.h>
unsigned int isqrt(unsigned int i)
```

Function

isqrt obtains the integral square root of the unsigned int *i*.

Return Value

isqrt returns the closest integer smaller or equal to the square root of *i*, expressed as an **unsigned int**.

Example

To use *isqrt* to check whether $n > 2$ is a prime number:

```
if (!(n & 01))
    return (NOTPRIME);
sq = isqrt(n);
for (div = 3; div <= sq; div += 2)
    if (!(n % div))
        return (NOTPRIME);
return (PRIME);
```

See Also

lsqrt, *sqrt*

Notes

isqrt is packaged in the integer library.

isspace

Description

Test for whitespace character

Syntax

```
#include <ctype.h>
int isspace(int c)
```

Function

isspace tests whether **c** is a whitespace character. Whitespace characters are horizontal tab (`'\t'`), newline (`'\n'`), vertical tab (`'\v'`), form feed (`'\f'`), carriage return (`'\r'`), and space (`' '`).

Return Value

isspace returns nonzero if **c** is a whitespace character; otherwise the value returned is zero.

Example

To skip leading whitespace:

```
while (isspace(*s))
    ++s;
```

See Also

isctrl, *isgraph*, *isprint*, *ispunct*

Notes

If the argument is outside the range $[-1, 255]$, the result is undefined.

isspace is packaged in the integer library.

isupper

Description

Test for uppercase character

Syntax

```
int isupper(int c)
```

Function

isupper tests whether **c** is an uppercase alphabetic character.

Return Value

isupper returns nonzero if **c** is an uppercase character; otherwise the value returned is zero.

Example

To convert to lowercase:

```
if (isupper(c))  
    c += 'a' - 'A'; /* also see tolower() */
```

See Also

isalnum, *isalpha*, *isdigit*, *islower*, *isxdigit*, *tolower*, *toupper*

Notes

If the argument is outside the range [-1, 255], the result is undefined.

isupper is packaged in the integer library.

isxdigit

Description

Test for hexadecimal digit

Syntax

```
#include <ctype.h>
int isxdigit(int c)
```

Function

isxdigit tests whether **c** is a hexadecimal digit, i.e. in the set [0123456789abcdefABCDEF].

Return Value

isxdigit returns nonzero if **c** is a hexadecimal digit; otherwise the value returned is zero.

Example

To accumulate a hexadecimal digit:

```
for (sum = 0; isxdigit(*s); ++s)
    if (isdigit(*s))
        sum = sum * 10 + *s - '0';
    else
        sum = sum * 10 + tolower(*s) + (10 - 'a');
```

See Also

isalnum, *isalpha*, *isdigit*, *islower*, *isupper*, *tolower*, *toupper*

Notes

If the argument is outside the range [-1, 255], the result is undefined.

isxdigit is packaged in the integer library.

labs

Description

Find long absolute value

Syntax

```
#include <stdlib.h>
long labs(long l)
```

Function

labs obtains the absolute value of **l**. No check is made to see that the result can be properly represented.

Return Value

labs returns the absolute value of **l**, expressed as an long int.

Example

To print out a debit or credit balance:

```
printf("balance %ld%s\n",labs(bal),(bal < 0) ? "CR" : "");
```

See Also

abs, fabs

Notes

labs is packaged in the integer library.

ldexp

Description

Scale double exponent

Syntax

```
#include <math.h>

double ldexp(double x, int exp)
```

Function

ldexp multiplies the double **x** by two raised to the integer power **exp**.

Return Value

ldexp returns the double result $x * (1 \ll \text{exp})$ expressed as a double floating value. If a range error occurs, *ldexp* returns **HUGE_VAL**.

Example

x	exp	ldexp(x, exp)
1.0	1	2.0
1.0	0	1.0
1.0	-1	0.5
0.0	0	0.0

See Also

frexp, *modf*

Notes

ldexp is packaged in the floating point library.

ldiv

Description

Long divide with quotient and remainder

Syntax

```
#include <stdlib.h>
ldiv_t ldiv(long numer, long denom)
```

Function

ldiv divides the long integer *numer* by the long integer *denom* and returns the quotient and the remainder in a structure of type *ldiv_t*. The field *quot* contains the quotient and the field *rem* contains the remainder.

Return Value

ldiv returns a structure of type *ldiv_t* containing both quotient and remainder.

Example

To get minutes and seconds from a delay in seconds:

```
ldiv_t result;
result = ldiv(time, 60L);
min = result.quot;
sec = result.rem;
```

See Also

div

Notes

ldiv is packaged in the integer library.

log

Description

Natural logarithm

Syntax

```
#include <math.h>
double log(double x)
```

Function

log computes the natural logarithm of **x** to full double precision.

Return Value

log returns the closest internal representation to $\log(x)$, expressed as a double floating value. If the input argument is less than zero, or is too large to be represented, *log* returns zero.

Example

To compute the hyperbolic arccosine of **x**:

```
arccosh = log(x + sqrt(x * x - 1));
```

See Also

exp

Notes

log is packaged in the floating point library.

log10

Description

Common logarithm

Syntax

```
#include <math.h>
double log10(double x)
```

Function

log10 computes the common log of **x** to full double precision by computing the natural log of **x** divided by the natural log of 10. If the input argument is less than zero, a domain error will occur. If the input argument is zero, a range error will occur.

Return Value

log10 returns the nearest internal representation to *log10 x*, expressed as a double floating value. If the input argument is less than or equal to zero, *log10* returns zero.

Example

To determine the number of digits in **x**, where **x** is a positive integer expressed as a double:

```
ndig = log10(x) + 1;
```

See Also

log

Notes

log10 is packaged in the floating point library.

longjmp

Description

Restore calling environment

Syntax

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val)
```

Function

longjmp restores the environment saved in *env* by *setjmp*. If *env* has not been set by a call to *setjmp*, or if the caller has returned in the meantime, the resulting behaviour is unpredictable.

All accessible objects have their values restored when *longjmp* is called, except for objects of storage class register, the values of which have been changed between the *setjmp* and *longjmp* calls.

Return Value

When *longjmp* returns, program execution continues as if the corresponding call to *setjmp* had returned the value *val*. *longjmp* cannot force *setjmp* to return the value zero. If *val* is zero, *setjmp* returns the value one.

Example

You can write a generic error handler as:

```
void handle(int err)
{
    extern jmp_buf env;
    longjmp(env, err); /* return from setjmp */
}
```

See Also

setjmp

Notes

longjmp is packaged in the integer library.

lsqrt

Description

Long integer square root

Syntax

```
#include <stdlib.h>
unsigned int lsqrt(unsigned long l)
```

Function

lsqrt obtains the integral square root of the unsigned long *l*.

Return Value

lsqrt returns the closest integer smaller or equal to the square root of *l*, expressed as an **unsigned int**.

Example

To use *lsqrt* to check whether $n > 2$ is a prime number:

```
if (!(n & 01))
    return (NOTPRIME);
sq = lsqrt(n);
for (div = 3; div <= sq; div += 2)
    if (!(n % div))
        return (NOTPRIME);
return (PRIME);
```

See Also

isqrt, *sqrt*

Notes

lsqrt is packaged in the integer library.

malloc

Description

Allocate space on the heap

Syntax

```
#include <stdlib.h>
void *malloc(unsigned int nbytes)
```

Function

malloc allocates space on the heap for an item of size *nbytes*. The space allocated is guaranteed to be at least *nbytes* long, starting from the pointer returned, which is guaranteed to be on a proper storage boundary for an object of any type. The heap is grown as necessary. If space is exhausted, *malloc* returns a null pointer.

Return Value

malloc returns a pointer to the start of the allocated cell if successful; otherwise it returns NULL. The pointer returned may be assigned to an object of any type without casting.

Example

To allocate an array of ten doubles:

```
double *pd;

pd = malloc(10 * sizeof *pd);
```

See Also

calloc, *free*, *realloc*

Notes

malloc is packaged in the integer library.

max

Description

Test for maximum

Syntax

```
#include <stdlib.h>
max(a,b)
```

Function

max obtains the maximum of its two arguments, **a** and **b**. Since *max* is implemented as a C preprocessor macro, its arguments can be any numerical type, and type coercion occurs automatically.

Return Value

max is a numerical rvalue of the form $((a < b) ? b : a)$, suitably parenthesized.

Example

To set a new maximum level:

```
hiwater = max(hiwater, level);
```

See Also

min

Notes

max is an extension to the proposed ANSI C standard.

max is a macro declared in the *<stdlib.h>* header file. You can use it by including *<stdlib.h>* with your program. Because it is a macro, *max* cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated other than once.

memchr

Description

Scan buffer for character

Syntax

```
#include <string.h>
void *memchr(void *s, int c, unsigned int n)
```

Function

memchr looks for the first occurrence of a specific character **c** in an **n** character buffer starting at **s**.

Return Value

memchr returns a pointer to the first character that matches **c**, or NULL if no character matches.

Example

To map *keybuf[]* characters into *subst[]* characters:

```
if ((t = memchr(keybuf, *s, KEYSIZ)) != NULL)
    *s = subst[t - keybuf];
```

See Also

strchr, *strcspn*, *strpbrk*, *strrchr*, *strspn*

Notes

memchr is packaged in the integer library.

memcmp

Description

Compare two buffers for lexical order

Syntax

```
#include <string.h>
int memcmp(void *s1, void *s2, unsigned int n)
```

Function

memcmp compares two text buffers, character by character, for lexical order in the character collating sequence. The first buffer starts at **s1**, the second at **s2**; both buffers are **n** characters long.

Return Value

memcmp returns a short integer greater than, equal to, or less than zero, according to whether **s1** is lexicographically greater than, equal to, or less than **s2**.

Example

To look for the string “*include*” in name:

```
if (memcmp(name, "include", 7) == 0)
    doinclude();
```

See Also

strcmp, *strncmp*

Notes

memcmp is packaged in the integer library.

memcpy

Description

Copy one buffer to another

Syntax

```
#include <string.h>
void *memcpy(void *s1, void *s2, unsigned int n)
```

Function

memcpy copies the first **n** characters starting at location **s2** into the buffer beginning at **s1**.

Return Value

memcpy returns **s1**.

Example

To place “*first string, second string*” in *buf[]*:

```
memcpy(buf, "first string", 12);
memcpy(buf + 13, ", second string", 15);
```

See Also

strcpy, *strncpy*

Notes

memcpy is packaged in the integer library.

memmove

Description

Copy one buffer to another

Syntax

```
#include <string.h>
void *memmove(void *s1, void *s2, unsigned int n)
```

Function

memmove copies the first **n** characters starting at location **s2** into the buffer beginning at **s1**. If the two buffers overlap, the function performs the copy in the appropriate sequence, so the copy is not corrupted.

Return Value

memmove returns **s1**.

Example

To shift an array of characters:

```
memmove(buf, &buf[5], 10);
```

See Also

memcpy

Notes

memmove is packaged in the integer library.

memset

Description

Propagate fill character throughout buffer

Syntax

```
#include <string.h>
void *memset(void *s, int c, unsigned int n)
```

Function

memset floods the **n** character buffer starting at **s** with fill character **c**.

Return Value

memset returns **s**.

Example

To flood a 512-byte buffer with NULs:

```
memset(buf, '\0', BUFSIZ);
```

Notes

memset is packaged in the integer library and may be implemented as an *inline* function.

min

Description

Test for minimum

Syntax

```
#include <stdlib.h>
min(a, b)
```

Function

min obtains the minimum of its two arguments, **a** and **b**. Since *min* is implemented as a C preprocessor macro, its arguments can be any numerical type, and type coercion occurs automatically.

Return Value

min is a numerical rvalue of the form $((a < b) ? a : b)$, suitably parenthesized.

Example

To set a new minimum level:

```
nmove = min(space, size);
```

See Also

max

Notes

min is an extension to the ANSI C standard.

min is a macro declared in the *<stdlib.h>* header file. You can use it by including *<stdlib.h>* with your program. Because it is a macro, *min* cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated more than once.

modf

Description

Extract fraction and integer from double

Syntax

```
#include <math.h>
double modf(double val, double *pd)
```

Function

modf partitions the double *val* into an integer portion, which is delivered to **pd*, and a fractional portion, which is returned as the value of the function. If the integer portion cannot be represented properly in an int, the result is truncated on the left without complaint.

Return Value

modf returns the signed fractional portion of *val* as a double floating value, and writes the integer portion at **pd*.

Example

val	*pd	modf(val, *pd)
5.1	5	0.1
5.0	5	0.0
4.9	4	0.9
0.0	0	0.0
-1.4	-1	-0.4

See Also

frexp, *ldexp*

Notes

modf is packaged in the floating point library.

`_par`

Description

Test or get the parity

Syntax

```
#include <processor.h>
@inline int _par(int)
```

Function

`_par` is an inline function allowing to test or get the *parity* of the argument expression, by using the PAR instruction. When used in an *if* construct, this function expands directly to a **bcc** or **bcs** instruction. When used in an expression, it expands in order to build in a register the value **0** or **1** depending on the *carry* bit value.

Return Value

`_par` returns **0** or **1** in a register if such a value is needed.

Example

<code>if (_par(x))</code>	produces	<code>ldw r2, (r1, #2)</code>
<code>even = 0;</code>		<code>par r2</code>
		<code>bcc L1</code>
		<code>stw r0, (r1, #4)</code>
	<code>L1:</code>	
 <code>even = _par(x);</code>	produces	 <code>ldw r2, (r1, #2)</code>
		<code>par r2</code>
		<code>adc r3, r0, r0</code>
		<code>stw r3, (r1, #4)</code>

Notes

`_par` is an inline function and then is not defined in any library. It is therefore not possible to take its address. For more information, see “[Inline Function](#)” in **Chapter 3**.

pow

Description

Raise *x* to the *y* power

Syntax

```
#include <math.h>
double pow(double x, double y)
```

Function

pow computes the value of *x* raised to the power of *y*.

Return Value

pow returns the value of *x* raised to the power of *y*, expressed as a double floating value. If *x* is zero and *y* is less than or equal to zero, or if *x* is negative and *y* is not an integer, *pow* returns zero.

Example

<i>x</i>	<i>y</i>	<i>pow(x, y)</i>
2.0	2.0	4.0
2.0	1.0	2.0
2.0	0.0	1.0
1.0	any	1.0
0.0	-2.0	0
-1.0	2.0	1.0
-1.0	2.1	0

See Also

exp

Notes

pow is packaged in the floating point library.

printf

Description

Output formatted arguments to stdout

Syntax

```
#include <stdio.h>
int printf(char *fmt, ...)
```

Function

printf writes formatted output to the output stream using the format string at *fmt* and the arguments specified by ..., as described below.

printf uses *putchar* to output each character.

Format Specifiers

The format string at *fmt* consists of literal text to be output, interspersed with conversion specifications that determine how the arguments are to be interpreted and how they are to be converted for output. If there are insufficient arguments for the format, the results are undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. *printf* returns when the end of the format string is encountered.

Each *<conversion specification>* is started by the character '%'. After the '%', the following appear in sequence:

<flags> - zero or more which modify the meaning of the conversion specification.

<field width> - a decimal number which optionally specifies a minimum field width. If the converted value has fewer characters than the field width, it is padded on the left (or right, if the left adjustment flag has been given) to the field width. The padding is with spaces unless the field width digit string starts with zero, in which case the padding is with zeros.

<precision> - a decimal number which specifies the minimum number of digits to appear for **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal point for **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of characters to be printed from a string in an **s** conversion. The precision takes the form of a period followed by a decimal digit string. A null digit string is treated as zero.

h - optionally specifies that the following **d**, **i**, **o**, **u**, **x**, or **X** conversion character applies to a short int or unsigned short int argument (the argument will have been widened according to the integral widening conversions, and its value must be cast to short or unsigned short before printing). It specifies a short pointer argument if associated with the **p** conversion character. If an **h** appears with any other conversion character, it is ignored.

l - optionally specifies that the **d**, **i**, **o**, **u**, **x**, and **X** conversion character applies to a long int or unsigned long int argument. It specifies a long or far pointer argument if used with the **p** conversion character. If the **l** appears with any other conversion character, it is ignored.

L - optionally specifies that the following **e**, **E**, **f**, **g**, and **G** conversion character applies to a long double argument. If the **L** appears with any other conversion character, it is ignored.

<conversion character> - character that indicates the type of conversion to be applied.

A field width or precision, or both, may be indicated by an asterisk ***** instead of a digit string. In this case, an int argument supplies the field width or precision. The arguments supplying field width must appear before the optional argument to be converted. A negative field width argument is taken as a **-** flag followed by a positive field width. A negative precision argument is taken as if it were missing.

The **<flags>** field is zero or more of the following:

space - a space will be prepended if the first character of a signed conversion is not a sign. This flag will be ignored if space and **+** flags are both specified.

- result is to be converted to an “alternate form”. For **c**, **d**, **i**, **s**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be zero. For **p**, **x** and **X** conversion, a non-zero result will have **Ox** or **OX** prepended to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will contain a decimal point, even if no digits follow the point. For **g** and **G** conversions, trailing zeros will not be removed from the result, as they normally are. For **p** conversion, it designates hexadecimal output.

+ - result of signed conversion will begin with a plus or minus sign.

- - result of conversion will be left justified within the field.

The *<conversion character>* is one of the following:

% - a ‘%’ is printed. No argument is converted.

c - the least significant byte of the int argument is converted to a character and printed.

d, **i**, **o**, **u**, **x**, **X** - the int argument is converted to signed decimal (**d** or **i**), unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** or **X**); the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** are used for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is **1**. The result of converting a zero value with precision of zero is no characters.

e, **E** - the double argument is converted in the style **[-]d.ddde+dd**, where there is one digit before the decimal point and the number of digits after it is equal to the precision. If the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits. However, if the magnitude to be printed is greater than or equal to **1E+100**, additional exponent digits will be printed as necessary.

f - the double argument is converted to decimal notation in the style **[-]ddd.ddd**, where the number of digits following the decimal point is

equal to the precision specification. If the precision is missing, it is taken as 6. If the precision is explicitly zero, no decimal point appears. If a decimal point appears, at least one digit appears before it.

g, G - the double argument is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted; style **e** will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeros are removed from the result; a decimal point appears only if it is followed by a digit.

n - the argument is taken to be an `int *` pointer to an integer into which is written the number of characters written to the output stream so far by this call to *printf*. No argument is converted.

p - the argument is taken to be a `void *` pointer to an object. The value of the pointer is converted to a sequence of printable characters, and printed as a hexadecimal number with the number of digits printed being determined by the field width.

s - the argument is taken to be a `char *` pointer to a string. Characters from the string are written up to, but not including, the terminating NUL, or until the number of characters indicated by the precision are written. If the precision is missing, it is taken to be arbitrarily large, so all characters before the first NUL are printed.

If the character after ‘%’ is not a valid conversion character, the behavior is undefined.

If any argument is or points to an aggregate (except for an array of characters using **%s** conversion or any pointer using **%p** conversion), unpredictable results will occur.

A nonexistent or small field width does not cause truncation of a field; if the result is wider than the field width, the field is expanded to contain the conversion result.

Return Value

printf returns the number of characters transmitted, or a negative number if a write error occurs.

Notes

A call with more conversion specifiers than argument variables will cause unpredictable results.

Example

To print **arg**, which is a double with the value **5100.53**:

```
printf("%8.2f\n", arg);  
printf("%*.*f\n", 8, 2, arg);
```

both forms will output: **05100.53**

See Also

sprintf

Notes

printf is packaged in both the integer library and the floating point library. The functionality of the integer only version of *printf* is a subset of the functionality of the floating point version. The integer only version cannot print or manipulate floating point numbers. If your programs call the integer only version of *printf*, the following conversion specifiers are invalid: **e**, **E**, **f**, **g** and **G**. The **L** modifier is also invalid.

If *printf* encounters an invalid conversion specifier, the invalid specifier is ignored and no special message is generated.

putchar

Description

Put a character to output stream

Syntax

```
#include <stdio.h>
int putchar(char c)
```

Function

putchar copies **c** to the user specified output stream.

You must rewrite *putchar* in either C or assembly language to provide an interface to the output mechanism to the C library.

Return Value

putchar returns **c**. If a write error occurs, *putchar* returns EOF.

Example

To copy input to output:

```
while ((c = getchar()) != EOF)
    putchar(c);
```

See Also

getchar

Notes

putchar is packaged in the integer library, and is by default using the first serial port **SCI 1**.

puts

Description

Put a text line to output stream

Syntax

```
#include <stdio.h>
int puts(char *s)
```

Function

puts copies characters from the buffer starting at **s** to the output stream and appends a newline character to the output stream.

puts uses *putchar* to output each character. The terminating NUL is not copied.

Return Value

puts returns zero if successful, or else nonzero if a write error occurs.

Example

To copy input to output, line by line:

```
while (puts(gets(buf)))
    ;
```

See Also

gets

Notes

puts is packaged in the integer library.

rand

Description

Generate pseudo-random number

Syntax

```
#include <stdlib.h>
int rand(void)
```

Function

rand computes successive pseudo-random integers in the range [0, 32767], using a linear multiplicative algorithm which has a period of 2 raised to the power of 32.

Example

```
int dice()
{
    return (rand() % 6 + 1);
}
```

Return Value

rand returns a pseudo-random integer.

See Also

srand

Notes

rand is packaged in the integer library.

realloc

Description

Reallocate space on the heap

Syntax

```
#include <stdlib.h>
void realloc(void *ptr, unsigned int nbytes)
```

Function

realloc grows or shrinks the size of the cell pointed to by *ptr* to the size specified by *nbytes*. The contents of the cell will be unchanged up to the lesser of the new and old sizes. The cell pointer *ptr* must have been obtained by an earlier *calloc*, *malloc*, or *realloc* call; otherwise the heap will become corrupted.

Return Value

realloc returns a pointer to the start of the possibly moved cell if successful. Otherwise *realloc* returns NULL and the cell and *ptr* are unchanged. The pointer returned may be assigned to an object of any type without casting.

Example

To adjust *p* to be *n* doubles in size:

```
p = realloc(p, n * sizeof(double));
```

See Also

calloc, *free*, *malloc*

Notes

realloc is packaged in the integer library.

sbreak

Description

Allocate new memory

Syntax

```
void *sbreak(unsigned int size)
```

Function

sbreak modifies the program memory allocation as necessary, to make available at least *size* contiguous bytes of new memory, on a storage boundary adequate for representing any type of data. There is no guarantee that successive calls to *sbreak* will deliver contiguous areas of memory.

Return Value

sbreak returns a pointer to the start of the new memory if successful; otherwise the value returned is NULL.

Example

To buy space for an array of symbols:

```
if (!(p = sbreak(nsyms * sizeof (symbol))))  
    remark("not enough memory!", NULL);
```

Notes

sbreak is packaged in the integer library.

sbreak is an extension to the ANSI C standard.

scanf

Description

Read formatted input

Syntax

```
#include <stdio.h>
int scanf(char *fmt, ...)
```

Function

scanf reads formatted input from the output stream using the format string at *fmt* and the arguments specified by ..., as described below.

scanf uses *getchar* to read each character.

The behavior is unpredictable if there are insufficient argument pointers for the format. If the format string is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored.

Format Specifiers

The format string may contain:

- any number of spaces, horizontal tabs, and newline characters which cause input to be read up to the next non-whitespace character, and
- ordinary characters other than '%' which must match the next character of the input stream.

Each *<conversion specification>*, the definition of which follows, consists of the character '%', an optional assignment-suppressing character '*', an optional maximum field width, an optional **h**, **l** or **L** indicating the size of the receiving object, and a *<conversion character>*, described below.

A conversion specification directs the conversion of the next input field. The result is placed in the object pointed to by the subsequent argument, unless assignment suppression was indicated by a '*'. An

input field is a string of non-space characters; it extends to the next conflicting character or until the field width, if specified, is exhausted.

The conversion specification indicates the interpretation of the input field; the corresponding pointer argument must be a restricted type. The *<conversion character>* is one of the following:

% - a single **%** is expected in the input at this point; no assignment occurs.

If the character after '**%**' is not a valid conversion character, the behavior is undefined.

c - a character is expected; the subsequent argument must be of type pointer to char. The normal behavior (skip over space characters) is suppressed in this case; to read the next non-space character, use **%1s**. If a field width is specified, the corresponding argument must refer to a character array; the indicated number of characters is read.

d - a decimal integer is expected; the subsequent argument must be a pointer to integer.

e, f, g - a float is expected; the subsequent argument must be a pointer to float. The input format for floating point numbers is an optionally signed sequence of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or **e**, followed by an optionally signed integer.

i - an integer is expected; the subsequent argument must be a pointer to integer. If the input field begins with the characters **0x** or **0X**, the field is taken as a hexadecimal integer. If the input field begins with the character **0**, the field is taken as an octal integer. Otherwise, the input field is taken as a decimal integer.

n - no input is consumed; the subsequent argument must be an *int* * pointer to an integer into which is written the number of characters read from the input stream so far by this call to *scanf*.

o - an octal integer is expected; the subsequent argument must be a pointer to integer.

p - a pointer is expected; the subsequent argument must be a *void ** pointer. The format of the input field should be the same as that produced by the **%p** conversion of *printf*. On any input other than a value printed earlier during the same program execution, the behavior of the **%p** conversion is undefined.

s - a character string is expected; the subsequent argument must be a *char ** pointer to an array large enough to hold the string and a terminating NUL, which will be added automatically. The input field is terminated by a space, a horizontal tab, or a newline, which is not part of the field.

u - an unsigned decimal integer is expected; the subsequent argument must be a pointer to integer.

x - a hexadecimal integer is expected; a subsequent argument must be a pointer to integer.

[- a string that is not to be delimited by spaces is expected; the subsequent argument must be a *char ** just as for **%s**. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not a circumflex '^', the input field consists of all characters up to the first character that is not in the set between the brackets; if the first character after the left bracket is a circumflex, the input field consists of all characters up to the first character that is in the set of the remaining characters between the brackets. A NUL character will be appended to the input.

The conversion characters **d**, **i**, **o**, **u** and **x** may be preceded by **l** to indicate that the subsequent argument is a pointer to long int rather than a pointer to int, or by **h** to indicate that it is a pointer to short int. Similarly, the conversion characters **e** and **f** may be preceded by **l** to indicate that the subsequent argument is a pointer to double rather than a pointer to float, or by **L** to indicate a pointer to long double.

The conversion characters **e**, **g** or **x** may be capitalized. However, the use of upper case has no effect on the conversion process and both upper and lower case input is accepted.

If conversion terminates on a conflicting input character, that character is left unread in the input stream. Trailing white space (including a newline) is left unread unless matched in the control string. The success of literal matches and suppressed assignments is not directly determinable other than via the **%n** conversion.

Return Value

scanf returns the number of assigned input items, which can be zero if there is an early conflict between an input character and the format, or EOF if end of file is encountered before the first conflict or conversion.

Example

To be certain of a dubious request:

```
printf("are you sure?");  
if (scanf("%c", &ans) && (ans == 'Y' || ans == 'y'))  
    scrog();
```

See Also

sscanf

Notes

scanf is packaged in both the integer library and the floating point library. The functionality of the integer only version of *scanf* is a subset of the functionality of the floating point version. The integer only version cannot read or manipulate floating point numbers. If your programs call the integer only version of *scanf*, the following conversion specifiers are invalid: **e**, **f**, **g** and **p**. The **L** flag is also invalid.

If an invalid conversion specifier is encountered, it is ignored.

setjmp

Description

Save calling environment

Syntax

```
#include <setjmp.h>
int setjmp(jmp_buf_env)
```

Function

setjmp saves the calling environment in *env* for later use by the *longjmp* function.

Since *setjmp* manipulates the stack, it should never be used except as the single operand in a switch statement.

Return Value

setjmp returns zero on its initial call, or the argument to a *longjmp* call that uses the same *env*.

Example

To call any event until it returns 0 or 1 and calls *longjmp*, which will then start execution at the function *event0* or *event1*:

```
static jmp_buf ev[2];
switch (setjmp(ev[0]))
{
case 0:          /* registered */
    break;
default:         /* event 0 occurred */
    event0();
    next();
}
switch (setjmp(ev[1]))
{
case 0:          /* registered */
    break;
default:         /* event 1 occurred */
    event1();
    next();
}
```



```
        next();  
        ...  
next()  
{  
    int i;  
  
    for (; ; )  
    {  
        i = anyevent();  
        if (i == 0 || i == 1)  
            longjmp(ev[i]);  
    }  
}
```

See Also*longjmp***Notes***setjmp* is packaged in the integer library.

`_sif`

Description

Set interrupt flag

Syntax

```
#include <processor.h>
@inline void _sif()
```

Function

`_sif` is an inline function allowing to set the interrupt flag by using the SIF instruction. If an argument is specified, it is evaluated and associated to a register SIF instruction.

Return Value

`_sif` returns nothing.

Example

<code>_sif();</code>	produces	<code>sif</code>
<code>sif(chan);</code>	produces	<code>ldw r2, (r1, #2)</code> <code>sif r2</code>

Notes

`_sif` is an inline function and then is not defined in any library. It is therefore not possible to take its address. For more information, see “[Inline Function](#)” in **Chapter 3**.

sin

Description

Sin

Syntax

```
#include <math.h>
double sin(double x)
```

Function

sin computes the sine of **x**, expressed in radians, to full double precision. If the magnitude of **x** is too large to contain a fractional quadrant part, the value of sin is 0.

Return Value

sin returns the closest internal representation to *sin(x)* in the range $[-\pi/2, \pi/2]$, expressed as a double floating value. A large argument may return a meaningless result.

Example

To rotate a vector through the angle *theta*:

```
xnew = xold * cos(theta) - yold * sin(theta);
ynew = xold * sin(theta) + yold * cos(theta);
```

See Also

cos, *tan*

Notes

sin is packaged in the floating point library.

sinh

Description

Hyperbolic sine

Syntax

```
#include <math.h>
double sinh(double x)
```

Function

sinh computes the hyperbolic sine of **x** to full double precision.

Return Value

sinh returns the closest internal representation to $\sinh(x)$, expressed as a double floating value. If the result is too large to be properly represented, *sinh* returns zero.

Example

To obtain the hyperbolic sine of complex **z**:

```
typedef struct
{
    double x, iy;
}complex;

complex z;

z.x = sinh(z.x) * cos(z.iy);
z.iy = cosh(z.x) * sin(z.iy);
```

See Also

cosh, *exp*, *tanh*

Notes

sinh is packaged in the floating point library.

sprintf

Description

Output arguments formatted to buffer

Syntax

```
#include <stdio.h>
int sprintf(char *s, char fmt, ...)
```

Function

sprintf writes formatted to the buffer pointed at by **s** using the format string at *fmt* and the arguments specified by ..., in exactly the same way as *printf*. See the description of the *printf* function for information on the format conversion specifiers. A NUL character is written after the last character in the buffer.

Return Value

sprintf returns the numbers of characters written, not including the terminating NUL character.

Example

To format a double at **d** into **buf**:

```
sprintf(buf, "%10f\n", d);
```

See Also

printf

Notes

sprintf is packaged in both the integer library and the floating point library. The functionality of the integer only version of *sprintf* is a subset of the functionality of the floating point version. The integer only version cannot print or manipulate floating point numbers. If your programs call the integer only version of *sprintf*, the following conversion specifiers are invalid: **e**, **E**, **f**, **g** and **G**. The **L** flag is also invalid.

sqrt

Description

Real square root

Syntax

```
#include <math.h>
double sqrt(double x)
```

Function

sqrt computes the square root of *x* to full double precision.

Return Value

sqrt returns the nearest internal representation to *sqrt(x)*, expressed as a double floating value. If *x* is negative, *sqrt* returns zero.

Example

To use *sqrt* to check whether $n > 2$ is a prime number:

```
if (!(n & 01))
    return (NOTPRIME);
sq = sqrt((double)n);
for (div = 3; div <= sq; div += 2)
    if (!(n % div))
        return (NOTPRIME);
return (PRIME);
```

See Also

isqrt, *lsqrt*

Notes

sqrt is packaged in the floating point library.

srand

Description

Seed pseudo-random number generator

Syntax

```
#include <stdlib.h>
void srand(unsigned char nseed)
```

Function

srand uses *nseed* as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to *rand*. If *srand* is called with the same seed value, the sequence of pseudo-random numbers will be repeated. The initial seed value used by *rand* and *srand* is 0.

Return Value

Nothing.

Example

To set up a new sequence of random numbers:

```
srand(103);
```

See Also

rand

Notes

srand is packaged in the integer library.

sscanf

Description

Read formatted input from a string

Syntax

```
#include <stdio.h>
int sscanf(char *s, char *fmt, ...)
```

Function

sscanf reads formatted input from the NUL-terminated string pointed at by **s** using the format string at *fmt* and the arguments specified by ..., in exactly the same way as *scanf*. See the description of the *scanf* function for information on the format conversion specifiers.

Return Value

sscanf returns the number of assigned input items, which can be zero if there is an early conflict between an input character and the format, or EOF if the end of the string is encountered before the first conflict or conversion.

See Also

scanf

Notes

sscanf is packaged in both the integer library and the floating point library. The functionality of the integer only version of *sscanf* is a subset of the functionality of the floating point version. The integer only version cannot print or manipulate floating point numbers. If your programs call the integer only version of *sscanf*, the following conversion specifiers are invalid: **e**, **f**, **g** and **p**. The **L** flag is also invalid.

`_ssem`

Description

Set a semaphore

Syntax

```
#include <processor.h>
@inline int _ssem(int)
```

Function

`_ssem` is an inline function allowing to set a semaphore given by the argument, by using the SSEM instruction, thus setting the carry according to the semaphore status. When used in an *if* construct, this function expands directly to a `bcc` or `bcs` instruction. When used in an expression, it expands in order to build in a register the value `0` or `1` depending on the *carry* bit value.

Return Value

`_ssem` returns `0` or `1` in the a register if such a value is needed.

Example

<code>if (_ssem(2))</code>	produces	<code>ssem #2</code>
<code> x = 0;</code>		<code>bcc L1</code>
		<code>stw r0, (r1, #2)</code>
		<code>L1:</code>
 <code>sem = _ssem(num);</code>	produces	 <code>ldw r2, (r1, #2)</code>
		<code>ssem r2</code>
		<code>adc r3, r0, r0</code>

Notes

`_ssem` is an inline function and then is not defined in any library. It is therefore not possible to take its address. For more information, see “[Inline Function](#)” in **Chapter 3**.

strcat

Description

Concatenate strings

Syntax

```
#include <string.h>
char *strcat(char *s1, char *s2)
```

Function

strcat appends a copy of the NUL terminated string at **s2** to the end of the NUL terminated string at **s1**. The first character of **s2** overlaps the NUL at the end of **s1**. A terminating NUL is always appended to **s1**.

Return Value

strcat returns **s1**.

Example

To place the strings “*first string, second string*” in *buf[]*:

```
buf[0] = '\0';
strcpy(buf, "first string");
strcat(buf, ", second string");
```

See Also

strncat

Notes

There is no way to specify the size of the destination area to prevent storage overwrites.

strcat is packaged in the integer library.

strchr

Description

Scan string for first occurrence of character

Syntax

```
#include <string.h>
char *strchr(char *s1, int c)
```

Function

strchr looks for the first occurrence of a specific character **c** in a NUL terminated target string **s**.

Return Value

strchr returns a pointer to the first character that matches **c**, or NULL if none does.

Example

To map *keystr*[] characters into *subst*[] characters:

```
if (t = strchr(keystr, *s))
    *s = subst[t - keystr];
```

See Also

memchr, *strcspn*, *strpbrk*, *strrchr*, *strspn*

Notes

strchr is packaged in the integer library.

strcmp

Description

Compare two strings for lexical order

Syntax

```
#include <string.h>
int strcmp(char *s1, char *s2)
```

Function

strcmp compares two text strings, character by character, for lexical order in the character collating sequence. The first string starts at **s1**, the second at **s2**. The strings must match, including their terminating NUL characters, in order for them to be equal.

Return Value

strcmp returns an integer greater than, equal to, or less than zero, according to whether **s1** is lexicographically greater than, equal to, or less than **s2**.

Example

To look for the string “include”:

```
if (strcmp(buf, "include") == 0)
    doinclude();
```

See Also

memcmp, *strncmp*

Notes

strcmp is packaged in the integer library.

strcpy

Description

Copy one string to another

Syntax

```
#include <string.h>
char *strcpy(char *s1, char *s2)
```

Function

strcpy copies the NUL terminated string at **s2** to the buffer pointed at by **s1**. The terminating NUL is also copied.

Return Value

strcpy returns **s1**.

Example

To make a copy of the string **s2** in *dest*:

```
strcpy(dest, s2);
```

See Also

memcpy, *strncpy*

Notes

There is no way to specify the size of the destination area, to prevent storage overwrites.

strcpy is packaged in the integer library, and may be implemented as an *inline* function.

strcspn

Description

Find the end of a span of characters in a set

Syntax

```
#include <string.h>
unsigned int strcspn(char *s1, char *s2)
```

Function

strcspn scans the string starting at **s1** for the first occurrence of a character in the string starting at **s2**. It computes a subscript **i** such that:

- **s1[i]** is a character in the string starting at **s1**
- **s1[i]** compares equal to some character in the string starting at **s2**, which may be its terminating null character.

Return Value

strcspn returns the lowest possible value of **i**. **s1[i]** designates the terminating null character if none of the characters in **s1** are in **s2**.

Example

To find the start of a decimal constant in a text string:

```
if (!str[i = strcspn(str, "0123456789+-")])
    printf("can't find number\n");
```

See Also

memchr, strchr, strpbrk, strrchr, strspn

Notes

strcspn is packaged in the integer library.

strlen

Description

Find length of a string

Syntax

```
#include <string.h>
unsigned int strlen(char *s)
```

Function

strlen scans the text string starting at **s** to determine the number of characters before the terminating NUL.

Return Value

The value returned is the number of characters in the string before the NUL.

Notes

strlen is packaged in the integer library and may be implemented as an *inline* function.

strncat

Description

Concatenate strings of length *n*

Syntax

```
#include <string.h>
char *strncat(char *s1, char *s2, unsigned int n)
```

Function

strncat appends a copy of the NUL terminated string at **s2** to the end of the NUL terminated string at **s1**. The first character of **s2** overlaps the NUL at the end of **s1**. **n** specifies the maximum number of characters to be copied, unless the terminating NUL in **s2** is encountered first. A terminating NUL is always appended to **s1**.

Return Value

strncat returns **s1**.

Example

To concatenate the strings “*day*” and “*light*”:

```
strcpy(s, "day");
strncat(s + 3, "light", 5);
```

See Also

strcat

Notes

strncat is packaged in the integer library.

strncmp

Description

Compare two *n* length strings for lexical order

Syntax

```
#include <string.h>
int strncmp(char *s1, char *s2, unsigned int n)
```

Function

strncmp compares two text strings, character by character, for lexical order in the character collating sequence. The first string starts at **s1**, the second at **s2**. **n** specifies the maximum number of characters to be compared, unless the terminating NUL in **s1** or **s2** is encountered first. The strings must match, including their terminating NUL character, in order for them to be equal.

Return Value

strncmp returns an integer greater than, equal to, or less than zero, according to whether **s1** is lexicographically greater than, equal to, or less than **s2**.

Example

To check for a particular error message:

```
if (strncmp(errmsg,
            "can't write output file", 23) == 0)
    cleanup(errmsg);
```

See Also

memcmp, *strcmp*

Notes

strncmp is packaged in the integer library.

strncpy

Description

Copy *n* length string

Syntax

```
#include <string.h>
char *strncpy(char *s1, char *s2, unsigned int n)
```

Function

strncpy copies the first *n* characters starting at location *s2* into the buffer beginning at *s1*. *n* specifies the maximum number of characters to be copied, unless the terminating NUL in *s2* is encountered first. In that case, additional NUL padding is appended to *s2* to copy a total of *n* characters.

Return Value

strncpy returns *s1*.

Example

To make a copy of the string *s2* in *dest*:

```
strncpy(dest, s2, n);
```

See Also

memcpy, *strcpy*

Notes

If the string *s2* points at is longer than *n* characters, the result may not be NUL-terminated.

strncpy is packaged in the integer library.

strupbrk

Description

Find occurrence in string of character in set

Syntax

```
#include <string.h>
char *strupbrk(char *s1, char *s2)
```

Function

strupbrk scans the NUL terminated string starting at **s1** for the first occurrence of a character in the NUL terminated set **s2**.

Return Value

strupbrk returns a pointer to the first character in **s1** that is also contained in the set **s2**, or a NULL if none does.

Example

To replace unprintable characters (as for a 64 character terminal):

```
while (string = strupbrk(string, "\\{||~"))
    *string = '@';
```

See Also

memchr, *strchr*, *strcspn*, *strrchr*, *strspn*

Notes

strupbrk is packaged in the integer library.

strchr

Description

Scan string for last occurrence of character

Syntax

```
#include <string.h>
char *strrchr(char *s,int c)
```

Function

strrchr looks for the last occurrence of a specific character **c** in a NUL terminated string starting at **s**.

Return Value

strrchr returns a pointer to the last character that matches **c**, or NULL if none does.

Example

To find a filename within a directory pathname:

```
if (s = strrchr("/usr/lib/libc.user", '/'))
    ++s;
```

See Also

memchr, *strchr*, *strpbrk*, *strcspn*, *strspn*

Notes

strrchr is packaged in the integer library.

strspn

Description

Find the end of a span of characters not in set

Syntax

```
#include <string.h>
unsigned int strspn(char *s1, char *s2)
```

Function

strspn scans the string starting at **s1** for the first occurrence of a character not in the string starting at **s2**. It computes a subscript **i** such that

- **s1[i]** is a character in the string starting at **s1**
- **s1[i]** compares equal to no character in the string starting at **s2**, except possibly its terminating null character.

Return Value

strspn returns the lowest possible value of **i**. **s1[i]** designates the terminating null character if all of the characters in **s1** are in **s2**.

Example

To check a string for characters other than decimal digits:

```
if (str[strspn(str, "0123456789")])
    printf("invalid number\n");
```

See Also

memchr, *strcspn*, *strchr*, *strpbrk*, *strrchr*

Notes

strspn is packaged in the integer library.

strstr

Description

Scan string for first occurrence of string

Syntax

```
#include <string.h>
char *strstr(char *s1, char *s2)
```

Function

strstr looks for the first occurrence of a specific string **s2** not including its terminating NUL, in a NUL terminated target string **s1**.

Return Value

strstr returns a pointer to the first character that matches **c**, or NULL if none does.

Example

To look for a keyword in a string:

```
if (t = strstr(buf, "LIST"))
    do_list(t);
```

See Also

memchr, *strcspn*, *strpbrk*, *strchr*, *strspn*

Notes

strstr is packaged in the integer library.

strtod

Description

Convert buffer to double

Syntax

```
#include <stdlib.h>
double strtod(char *nptr, char **endptr)
```

Function

strtod converts the string at *nptr* into a double. The string is taken as the text representation of a decimal number, with an optional fraction and exponent. Leading whitespace is skipped and an optional sign is permitted; conversion stops on the first unrecognizable character. Acceptable inputs match the pattern:

$$[+|-]d*[.d*][e[+|-]dd*]$$

where **d** is any decimal digit and **e** is the character ‘e’ or ‘E’. If *endptr* is not a null pointer, **endptr* is set to the address of the first unconverted character remaining in the string *nptr*. No checks are made against overflow, underflow, or invalid character strings.

Return Value

strtod returns the converted double value. If the string has no recognizable characters, it returns zero.

Example

To read a string from STDIN and convert it to a double at **d**:

```
gets(buf);
d = strtod(buf, NULL);
```

See Also

atoi, *atol*, *strtol*, *strtoul*

Notes

strtod is packaged in the floating point library.

strtol

Description

Convert buffer to long

Syntax

```
#include <stdlib.h>
long strtol(char *nptr, char **endptr, int base)
```

Function

strtol converts the string at *nptr* into a long integer. Leading whitespace is skipped and an optional sign is permitted; conversion stops on the first unrecognizable character. If base is not zero, characters **a-z** or **A-Z** represents digits in range 10-36. If base is zero, a leading “**0x**” or “**0X**” in the string indicates hexadecimal, a leading “**0**” indicates octal, otherwise the string is take as a decimal representation. If base is 16 and a leading “**0x**” or “**0X**” is present, it is skipped before to convert. If *endptr* is not a null pointer, **endptr* is set to the address of the first unconverted character in the string *nptr*.

No checks are made against overflow or invalid character strings.

Return Value

strtol returns the converted long integer. If the string has no recognizable characters, zero is returned.

Example

To read a string from STDIN and convert it to a long **l**:

```
gets(buf);
l = strtol(buf, NULL, 0);
```

See Also

atof, *atoi*, *strtoul*, *strtod*

Notes

strtol is packaged in the integer library.

strtoul

Description

Convert buffer to unsigned long

Syntax

```
#include <stdlib.h>
unsigned long strtoul(char *nptr, char **endptr,
                     int base)
```

Function

strtoul converts the string at *nptr* into a long integer. Leading whitespace is skipped and an optional sign is permitted; conversion stops on the first unrecognizable character. If base is not zero, characters **a-z** or **A-Z** represents digits in range 10-36. If base is zero, a leading “**0x**” or “**0X**” in the string indicates hexadecimal, a leading “**0**” indicates octal, otherwise the string is take as a decimal representation. If base is 16 and a leading “**0x**” or “**0X**” is present, it is skipped before to convert. If *endptr* is not a null pointer, **endptr* is set to the address of the first unconverted character in the string *nptr*.

No checks are made against overflow or invalid character strings.

Return Value

strtoul returns the converted long integer. If the string has no recognizable characters, zero is returned.

Example

To read a string from STDIN and convert it to a long l:

```
gets(buf);
l = strtoul(buf, NULL, 0);
```

See Also

atof, *atoi*, *strtol*, *strtod*

Notes

strtoul is a macro redefined to *strtol*.

tan

Description

Tangent

Syntax

```
#include <math.h>
double tan(double x)
```

Function

tan computes the tangent of **x**, expressed in radians, to full double precision.

Return Value

tan returns the nearest internal representation to *tan(x)*, in the range $[-\pi/2, \pi/2]$, expressed as a double floating value. If the number in **x** is too large to be represented, **tan** returns zero. An argument with a large size may return a meaningless value, *i.e.* when **x**/**(2 * pi)** has no fraction bits.

Example

To compute the tangent of *theta*:

```
y = tan(theta);
```

See Also

cos, *sin*

Notes

tan is packaged in the floating point library.

tanh

Description

Hyperbolic tangent

Syntax

```
#include <math.h>
double tanh(double x)
```

Function

tanh computes the value of the hyperbolic tangent of *x* to double precision.

Return Value

tanh returns the nearest internal representation to $\tanh(x)$, expressed as a double floating value. If the result is too large to be properly represented, *tanh* returns zero.

Example

To compute the hyperbolic tangent of *x*:

```
y = tanh(x);
```

See Also

cosh, *exp*, *sinh*

Notes

tanh is packaged in the floating point library.

tolower

Description

Convert character to lowercase if necessary

Syntax

```
#include <ctype.h>
int tolower(int c)
```

Function

tolower converts an uppercase letter to its lowercase equivalent, leaving all other characters unmodified.

Return Value

tolower returns the corresponding lowercase letter, or the unchanged character.

Example

To accumulate a hexadecimal digit:

```
for (sum = 0; isxdigit(*s); ++s)
    if (isdigit(*s)
        sum = sum * 16 + *s - '0';
    else
        sum = sum * 16 + tolower(*s) + (10 - 'a');
```

See Also

toupper

Notes

tolower is packaged in the integer library.

toupper

Description

Convert character to uppercase if necessary

Syntax

```
#include <ctype.h>
int toupper(int c)
```

Function

toupper converts a lowercase letter to its uppercase equivalent, leaving all other characters unmodified.

Return Value

toupper returns the corresponding uppercase letter, or the unchanged character.

Example

To convert a character string to uppercase letters:

```
for (i = 0; i < size; ++i)
    buf[i] = toupper(buf[i]);
```

See Also

tolower

Notes

toupper is packaged in the integer library.

va_arg

Description

Get pointer to next argument in list

Syntax

```
#include <stdarg.h>
type va_arg(va_list ap, type)
```

Function

The macro **va_arg** is an *rvalue* that computes the value of the next argument in a variable length argument list. Information on the argument list is stored in the array data object *ap*. You must first initialize *ap* with the macro *va_start*, and compute all earlier arguments in the list by expanding *va_arg* for each argument.

The type of the next argument is given by the type name *type*. The type name must be the same as the type of the next argument. Remember that the compiler widens an arithmetic argument to int, and converts an argument of type float to double. You write the type after conversion. Write int instead of char and double instead of float.

Do not write a type name that contains any parentheses. Use a type definition, if necessary, as in:

```
typedef int (*pfi)();
/* pointer to function returning int */
...
fun_ptr = va_arg(ap, pfi);
/* get function pointer argument */
```

Return Value

va_arg expands to an *rvalue* of type *type*. Its value is the value of the next argument. It alters the information stored in *ap* so that the next expansion of *va_arg* accesses the argument following.

Example

To write multiple strings to a file:

```
#include <stdio.h>
#include <stdarg.h>

main()
{
    void strput();
    strput(pf, "This is one string\n", \
           "and this is another...\n", (char *)0);
}

void strput(FILE *pf, ...);
void strput(char *ptr, ...)
void strput(ptr)
    char *ptr;
    {
        char ptr;
        va_list va;

        if (!ptr)
            return;
        else
            {
                puts(ptr);
                va_start(va, ptr);
                while (ptr = va_arg(va, char *))
                    puts(ptr);
                va_end(va);
            }
    }
```

See Also

va_end, *va_start*

Notes

va_arg is a macro declared in the *<stdarg.h>* header file. You can use it with any function that accepts a variable number of arguments, by including *<stdarg.h>* with your program.

va_end

Description

Stop accessing values in an argument list

Syntax

```
#include <stdarg.h>
void va_end(va_list ap)
```

Function

va_end is a macro which you must expand if you expand the macro *va_start* within a function that contains a variable length argument list. Information on the argument list is stored in the data object designated by *ap*. Designate the same data object in both *va_start* and *va_end*.

You expand *va_end* after you have accessed all argument values with the macro *va_arg*, before your program returns from the function that contains the variable length argument list. After you expand *va_end*, do not expand *va_arg* with the same *ap*. You need not expand *va_arg* within the function that contains the variable length argument list.

You must write an expansion of *va_end* as an expression statement containing a function call. The call must be followed by a semicolon.

Return Value

Nothing. *va_end* expands to a statement, not an expression.

Example

To write multiple strings to a file:

```
#include <stdio.h>
#include <stdarg.h>

main()
{
    void strput();

    strput(pf, "This is one string\n", \
          "and this is another...\n", (char *)0);
}
```



```
void strput(FILE *pf, ...);
void strput(char *ptr, ...)
void strput(ptr)
    char *ptr;
    {
        char ptr;
        va_list va;

        if (!ptr)
            return;
        else
            {
                puts(ptr);
                va_start(va, ptr);
                while (ptr = va_arg(va, char *))
                    puts(ptr);
                va_end(va);
            }
    }
```

See Also

va_arg, *va_start*

Notes

va_end is a macro declared in the `<stdarg.h>` header file. You can use it with any function that accepts a variable number of arguments, by including `<stdarg.h>` with your program.

va_start

Description

Start accessing values in an argument list

Syntax

```
#include <stdarg.h>
void va_start(va_list ap, parmN)
```

Function

va_start is a macro which you must expand before you expand the macro *va_arg*. It initializes the information stored in the data object designated by *ap*. The argument *parmN* must be the identifier you declare as the name of the last specified argument in the variable length argument list for the function. In the function prototype for the function, *parmN* is the argument name you write just before the , ...

The type of *parmN* must be one of the types assumed by an argument passed in the absence of a prototype. Its type must not be float or char. Also, *parmN* cannot have storage class register.

If you expand *va_start*, you must expand the macro *va_end* before your program returns from the function containing the variable length argument list.

You must write an expansion of *va_start* as an expression statement containing a function call. The call must be followed by a semicolon.

Return Value

Nothing. *va_start* expands to a statement, not an expression.

Example

To write multiple strings to a file:

```
#include <stdio.h>
#include <stdarg.h>

main()
{
```

```
void strput();
strput(pf, "This is one string\n", \
        "and this is another...\n", (char *)0);
}

void strput(FILE *pf, ...);
void strput(char *ptr, ...)
void strput(ptr)
    char *ptr;
    {
        char ptr;
        va_list va;

        if (!ptr)
            return;
        else
            {
                puts(ptr);
                va_start(va, ptr);
                while (ptr = va_arg(va, char *))
                    puts(ptr);
                va_end(va);
            }
    }
```

See Also

va_arg, *va_end*

Notes

va_start is a macro declared in the `<stdarg.h>` header file. You can use it with any function that accepts a variable number of arguments, by including `<stdarg.h>` with your program.

vprintf

Description

Output arguments formatted to stdout

Syntax

```
#include <stdio.h>
#include <stdarg.h>
int vsprintf(char*s, char fmt, va_list ap)
```

Function

vprintf writes formatted to the output stream using the format string at *fmt* and the arguments specified by pointer *ap*, in exactly the same way as *printf*. See the description of the *printf* function for information on the format conversion specifiers. The *va_start* macro must be executed before to call the *vprintf* function.

vprintf uses *putchar* to output each character.

Return Value

vprintf returns the numbers of characters transmitted.

Example

To format a double at *d* into *buf*:

```
va_start(aptr, fmt);
vprintf(fmt, aptr);
```

See Also

printf, *vsprintf*

Notes

vprintf is packaged in both the integer library and the floating point library. The functionality of the integer only version of *vprintf* is a subset of the functionality of the floating point version. The integer only version cannot print floating point numbers. If your programs call the integer only version of *vprintf*, the following conversion specifiers are invalid: **e**, **E**, **f**, **g** and **G**. The **L** flag is also invalid.

vsprintf

Description

Output arguments formatted to buffer

Syntax

```
#include <stdio.h>
#include <stdarg.h>
int vsprintf(char*s, char fmt, va_list ap)
```

Function

vsprintf writes formatted to the buffer pointed at by **s** using the format string at **fmt** and the arguments specified by pointer **ap**, in exactly the same way as **printf**. See the description of the **printf** function for information on the format conversion specifiers. A NUL character is written after the last character in the buffer. The **va_start** macro must be executed before to call the **vsprintf** function.

Return Value

vsprintf returns the numbers of characters written, not including the terminating NUL character.

Example

To format a double at **d** into **buf**:

```
va_start(aptr, fmt);
vsprintf(buf, fmt, aptr);
```

See Also

printf, *vprintf*

Notes

vsprintf is packaged in both the integer library and the floating point library. The functionality of the integer only version of **vsprintf** is a subset of the functionality of the floating point version. The integer only version cannot print floating point numbers. If your programs call the integer only version of **vsprintf**, the following conversion specifiers are invalid: **e**, **E**, **f**, **g** and **G**. The **L** flag is also invalid.

CHAPTER 5

Using The Assembler

The **caxgate** cross assembler translates your assembly language source files into relocatable object files. The C cross compiler calls *caxgate* to assemble your code automatically, unless specified otherwise. *caxgate* generates also listings if requested. This chapter includes the following sections:

- Invoking caxgate
- Object File
- Listings
- Assembly Language Syntax
- Branch Optimization
- C Style Directives
- Assembler Directives

Invoking *caxgate*

caxgate accepts the following command line options, each of which is described in detail below:

```
caxgate [options] <files>
-a    absolute assembler
-b    do not optimizes branches
-c    output cross reference
-d*>  define symbol=value
+e*   error file name
-ff   use formfeed in listing
-ft   force title in listing
-f#   fill byte value
-h*   include header
-i*>  include path
-l    output a listing
+l*   listing file name
-m    accept old syntax
-mi   accept label syntax
-o*   output file name
-pe   all equates public
-pl   keep local symbol
-p    all symbols public
-u    undefined in listing
-v    be verbose
-x    include line debug info
-xp   no path in debug info
-xx   include full debug info
```

- a** map all sections to absolute, including the predefined ones.
- b** do not optimize branch instructions. By default, the assembler replaces long branches by short branches wherever a shorter instruction can be used, and short branches by long branches wherever the displacement is too large. This optimization also applies to jump and jump to subroutines instructions.
- c** produce cross-reference information. The cross-reference information will be added at the end of the listing file; this option enforces the **-l** option.

- d*>** where * has the form **name=value**, defines **name** to have the value specified by **value**. This option is equivalent to using an **equ** directive in each of the source files.
- +e*** log errors from assembler in the text file * instead of displaying the messages on the terminal screen.
- ff** use *formfeed* character to skip pages in listing instead of using blank lines.
- ft** output a title in listing (date, file name, page). By default, no title is output.
- f#** define the value of the filling byte used to fill any gap created by the assembler directives. Default is 0.
- h*** include the file specified by * before starting assembly. It is equivalent to an **include** directive in each source file.
- i*>** define a path to be used by the **include** directive. Up to 20 paths can be defined. A path is a directory name and is **not** ended by any directory separator character.
- l** create a listing file. The name of the listing file is derived from the input file name by replacing the suffix by the *‘.ls’* extension.
- +l*** create a listing file in the text file *. If both **-l** and **+l** are specified, the listing file name is given by the **+l** option.
- m** accept the old Motorola syntax.
- mi** accept label that is not ended with a *‘:’* character.
- o*** write object code to the file *. If no file name is specified, the output file name is derived from the input file name, by replacing the rightmost extension in the input file name with the character *‘o’*. For example, if the input file name is *prog.s*, the default output file name is *prog.o*.

- pe** mark all symbols defined by an **equ** directive as **public**. This option has the same effect than adding a **xdef** directive for each of those symbols.
- pl** put locals in the symbol table. They are not published as externals and will be only displayed in the linker map file.
- p** mark all defined symbols as **public**. This option has the same effect than adding a **xdef** directive for each label.
- u** produce an error message in the listing file for all occurrence of an undefined symbol. This option enforces the **-l** option.
- v** display the name of each file which is processed.
- x** add line debug information to the object file.
- xp** do not prefix filenames in the debug information with any absolute path name. Debuggers will have to be informed about the actual files location.
- xx** add debug information in the object file for any label defining code or data. This option disables the **-p** option as only public or used labels are selected.

Each source file specified by *<files>* will be assembled separately, and will produce separate object and listing files. For each source file, if no errors are detected, *caxgate* generates an object file. If requested by the **-l** or **-c** options, *caxgate* generates a listing file even if errors are detected. Such lines are followed by an error message in the listing.

Object File

The object file produced by the assembler is a relocatable object in a format suitable for the linker *clnk*. This will normally consist of machine code, initialized data and relocation information. The object file also contains information about the sections used, a symbol table, and a debug symbol table.

Listings

The listing stream contains the source code used as input to the assembler, together with the hexadecimal representation of the corresponding object code and the address for which it was generated. The contents of the listing stream depends on the occurrence of the **list**, **nolist**, **clist**, **dlist** and **mlist** directives in the source. The format of the output is as follows:

<address> <generated_code> <source_line>

where *<address>* is the hexadecimal relocatable address where the *<source_line>* has been assembled, *<generated_code>* is the hexadecimal representation of the object code generated by the assembler and *<source_line>* is the original source line input to the assembler. If expansion of data, macros and included files is not enabled, the *<generated_code>* print will not contain a complete listing of all generated code.

Addresses in the listing output are the offsets from the start of the current section. After the linker has been executed, the listing files may be updated to contain absolute information by the **clabs** utility. Addresses and code will be updated to reflect the actual values as built by the linker.

Several directives are available to modify the listing display, such as **title** for the page header, **plen** for the page length, **page** for starting a new page, **tabs** for the tabulation characters expansion. By default, the listing file is not paginated. Pagination is enabled by using at least one **title** directive in the source file, or by specifying the **-ft** option on the command line. Otherwise, the **plen** and **page** directives are simply ignored. Some other directives such as **clist**, **mlist** or **dlist** control the amount of information produced in the listing.

A **cross-reference** table will be appended to the listing if the **-c** option has been specified. This table gives for each symbol its value, its attributes, the line number of the line where it has been defined, and the list of line numbers where it is referenced.

Assembly Language Syntax

The assembler *caxgate* conforms to the Motorola syntax as described in the document *Assembly Language Input Standard*. The assembly language consists of lines of text in the form:

```
[label:] [command [operands]] [; comment]
or
; comment
```

where ‘:’ indicates the end of a label and ‘;’ defines the start of a comment. The end of a line terminates a comment. The *command* field may be an **instruction**, a **directive** or a **macro call**.

Instruction mnemonics and assembler directives may be written in upper or lower case. The C compiler generates lowercase assembly language.

A source file must end with the **end** directive. All the following lines will be ignored by the assembler. If an **end** directive is found in an included file, it stops only the process for the included file.

Instructions

caxgate recognizes the following instructions:

adc	bfext	blo	cpch	or	stw
add	bffo	bls	csem	orh	sub
addh	bfins	blt	cs1	orl	subh
addl	bfinsi	bmi	csr	par	subl
and	bfinsx	bne	jal	rol	tfr
andh	bge	bpl	ldb	ror	xnor
andl	bgt	bra	ldh	rts	xnorh
asl	bhi	brk	ldl	sbc	xnorl
asr	bhs	bvc	ldw	sex	
bcc	bith	bvs	lsl	sif	
bcs	bitl	cmp	lsr	ssem	
beq	ble	cmpl	nop	stb	

The **operand** field of an instruction uses an **addressing** mode to describe the instruction argument. The following example demonstrates the accepted syntax:

```

sif                ; implicit
csem    #1         ; immediate
csem    r2         ; register
add     r1,r2,r3   ; register
ldw     r2,(r3)    ; indexed
ldw     r2,(r3,#2) ; indexed
ldw     r2,(r3,r4) ; indexed
ldw     r2,(r3+)   ; indexed
ldw     r2,(r3,-r4) ; indexed
bne     loop       ; relative

```

The assembler chooses the smallest addressing mode where several solutions are possible. It also allows 16-bit constants to be used on 16-bit instructions (add, and, or ...) and expands them in one or two instructions operating selectively on the high and/or low byte.

For an exact description of the above instructions, refer to the Motorola's *XGATE Reference Manual*.

Labels

A source line may begin with a **label**. Some directives require a label on the same line, otherwise this field is optional. A label must begin with an alphabetic character, the underscore character '_' or the period character '.'. It is continued by alphabetic (A-Z or a-z) or numeric (0,9) characters, underscores, dollar signs (\$) or periods. Labels are case sensitive. The processor register names are reserved and cannot be used as labels.

```

data1:dc.b    $56
c_reg:ds.b    1

```

When a label is used within a macro, it may be expanded more than once and in that case, the assembler will fail with a *multiply defined symbol* error. In order to avoid that problem, the special sequence '@' may be used as a label prefix. This sequence will be replaced by a unique sequence for each macro expansion. This prefix is only allowed inside a macro definition.

```

semwait:macro
\@loop:
    ssem    \1
    bcc     @loop
endm

```

Temporary Labels

The assembler allows temporary labels to be defined when there is no need to give them an explicit name. Such a label is composed by a decimal number immediately followed by a ‘\$’ character. Such a label is valid until the next standard label or the *local* directive. Then the same temporary label may be redefined without getting a multiply defined error message.

```
1$:  sub    r1,#1
      bne    1$
2$:  sub    r2,#1
      bne    2$
```

Temporary labels do not appear in the symbol table or the cross reference list.

For example, to define 3 different local blocks and create and use 3 different local labels named 10\$:

```
function1:
10$:  sub    r2,#1
      bne    10$
      add    r3,#1
      local
10$:  sub    r2,#1
      bne    10$
      add    r3,#1
      rts
function2:
10$:  sub    r2,#1
      bne    10$
      add    r3,#1
      rts
```

Constants

The assembler accepts **numeric** constants and **string** constants. *Numeric* constants are expressed in different bases depending on a *prefix* character as follows:

Number	Base
10	decimal (no prefix)
%1010	binary
@12	octal
\$A	hexadecimal

The assembler also accepts numerics constants in different bases depending on a *suffix* character as follow:

Suffix	Base
D, d or none	decimal (no prefix)
B or b	binary
Q or q	octal
0AH or 0Ah	hexadecimal

The *suffix* letter can be entered uppercase or lowercase. Hexadecimal numbers still **need** to start with a digit.

String constants are a series of printable characters between single or double quote characters:

```
'This is a string'
"This is also a string"
```

Depending on the context, a string constant will be seen either as a series of bytes, for a data initialization, or as a numeric; in which case, the string constant should be reduced to only one character.

```
hexa: dc.b    '0123456789ABCDEF'
start:cmpl   r2,#'A'      ; ASCII value of 'A'
```

Expressions

An expression consists of a number of labels and constants connected together by operators. Expressions are evaluated to 32-bit precision. Note that operators have the same precedence than in the C language.

A special label written ‘*’ is used to represent the current location address. Note that when ‘*’ is used as the operand of an instruction, it has the value of the program counter **before** code generation for that instruction. The set of accepted operators is:

+	addition
-	subtraction (negation)
*	multiplication
/	division
%	remainder (modulus)
&	bitwise and
	bitwise or
^	bitwise exclusive or
~	bitwise complement
<<	left shift
>>	right shift
==	equality
!=	difference
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
&&	logical and
	logical or
!	logical complement

These operators may be applied to constants without restrictions, but are restricted when applied to *relocatable* labels. For those labels, the **addition** and **subtraction** operators only are accepted and only in the following cases:

```
label + constant
label - constant
label1 - label2
```

NOTE

The difference of two relocatable labels is valid only if both symbols are not external symbols, and are defined in the same section.

An expression may also be constructed with a special operator. These expressions cannot be used with the previous operators and have to be specified alone.

high(expression)	upper byte
low(expression)	lower byte
page(expression)	page byte

These special operators evaluate an **expression** and extract the appropriate information from the result. The expression may be relocatable, and may use the set of operators if allowed.

high - extract the upper byte of the 16-bit expression

low - extract the lower byte of the 16-bit expression

page - extract the *page* value of the expression. It is computed by the linker according to the **-bs** option used. This is used to get the address extension when bank switching is used.

Macro Instructions

A **macro** instruction is a list of assembler commands collected under a unique name. This name becomes a new command for the following of the program. A **macro** begins with a **macro** directive and ends with a **endm** directive. All the lines between these two directives are recorded and associated with the macro name specified with the **macro** directive.

```
signex:macro                                ; sign extension
    ldw    r2,r0                          ; prepare MSW
    cmp    r3,#0                          ; test sign
    bpl    \@pos                          ; if not positive
    sub    r2,#1                          ; invert MSW
\@pos:
    endm                                ; end of macro
```

This macro is named *signex* and contains the code needed to perform a sign extension of **r3** into **r2**. Whenever needed, this macro can be expanded just by using its name in place of a standard instruction:

```
ldw    r3,(r1,#4)    ; load LSW
signex                                ; expand macro
stw    r2,(r1,#2)    ; store result
```

The resulting code will be the same as if the following code had been written:

```

ldw    r3, (r1, #4)    ; load LSW
ldw    r2, r0          ; prepare MSW
cmp     r3, #0          ; test sign
bpl     pos            ; if not positive
sub     r2, #1          ; invert MSW
pos:    stw    r2, (r1, #2) ; store result

```

A **macro** may have up to 35 *parameters*. A *parameter* is written **\1**, **\2**,... **\9**, **\A**,...**\Z** inside the macro body and refers explicitly to the first, second,... ninth *argument* and **\A** to **\Z** to denote the tenth to 35th operand on the invocation line, which are placed after the macro name, and separated by commas. Each *argument* replaces each occurrence of its corresponding *parameter*. An *argument* may be expressed as a **string** constant if it contains a comma character.

A macro can also handle named arguments instead of numbered argument. In such a case, the macro directive is followed by a list of argument named, each prefixed by a **** character, and separated by commas. Inside the macro body, arguments will be specified using the same syntax or a sequence starting by a **** character followed by the argument named placed between parenthesis. This alternate syntax is useful to concatenate the argument with a text string immediately starting with alphanumeric characters.

The special *parameter* **\#** is replaced by a numeric value corresponding to the number of *arguments* actually found on the invocation line.

In order to operate directly in memory, the previous macro may have been written using **numbered** syntax:

```

signex:macro                                ; sign extension
    ldw    \1, r0                          ; prepare MSW
    cmp     \2, #0                          ; test sign
    bpl     \@pos                          ; if not positive
    sub     \1, #1                          ; invert MSW
\@pos:
    endm                                ; end of macro

```

And called:

```

signex r2, r3                                ; sign extend word

```

This macro may also be written using the **named** syntax:

```
signex:macro \out,\in      ; sign extension
    ldw    \out,r0        ; prepare MSW
    cmp     \in,#0         ; test sign
    bpl     \@pos          ; if not positive
    sub     \out,#1        ; invert MSW
\@pos:
    endm                  ; end of macro
```

The form of a macro call is:

<code><name>[.<ext>] [<arguments>]</code>

The special parameter `\0` corresponds to an extension `<ext>` which may follow the macro name, separated by the period character `'.'`. An extension is a single letter which may represent the size of the operands and the result. For example:

```
table:macro
    dc.\0    1,2,3,4
endm
```

When invoking the macro:

```
table.b
```

will generate a table of byte:

```
dc.b    1,2,3,4
```

When invoking the macro:

```
table.w
```

will generate a table of word:

```
dc.w    1,2,3,4
```

The special parameter `*` is replaced by a sequence containing the list of all the passed arguments separated by commas. This syntax is useful to pass all the macro arguments to another macro or a **repeatl** directive.

The directive **mexit** may be used at any time to stop the macro expansion. It is generally used in conjunction with a conditional directive.

A macro call may be used within another macro definition. A macro definition cannot contain another macro definition.

If a listing is produced, the macro expansion lines are printed if enabled by the **mlist** directive. If enabled, the invocation line is not printed, and all the expanded lines are printed with all the *parameters* replaced by their corresponding *arguments*. Otherwise, the invocation line only is printed.

Conditional Directives

A **conditional directive** allows parts of the program to be assembled or not depending on a specific condition expressed in an **if** directive. The condition is an expression following the **if** command. The expression cannot be relocatable, and shall evaluate to a numeric result. If the condition is *false* (expression evaluated to zero), the lines following the **if** directive are skipped until an **endif** or **else** directive. Otherwise, the lines are normally assembled. If an **else** directive is encountered, the condition status is reversed, and the conditional process continues until the next **endif** directive.

```
if      debug == 1
ldw     r2,#message
jal     r6
endif
```

If the symbol **debug** is equal to **1**, the next two lines are assembled. Otherwise they are skipped.

```
if      offset > 255    ; if offset too large
addptr  offset          ; call a macro
else                    ; otherwise
addl    r2,#offset      ; adjust R2 register
endif
```

If the symbol *offset* is larger than 255, the macro **addptr** is expanded with *offset* as argument, otherwise the **addl** instruction is directly assembled.

Conditional directives may be nested. An **else** directive refers to the closest previous **if** directive, and an **endif** directive refers to the closest previous **if** or **else** directive.

If a listing is produced, the skipped lines are printed only if enabled by the **clist** directive. Otherwise, only the assembled lines are printed.

Sections

The assembler allows code and data to be splitted in **sections**. A *section* is a set of code or data referenced by a section name, and providing a contiguous block of relocatable information. A *section* is defined with a *section* directive, which creates a new section and redirects the following code and data thereto. The directive **switch** can be used to redirect the following code and data to another *section*.

```
xdata:      section          ; defines data section
xtext:      section          ; defines text section
start:
    ldw     r2,#value        ; fills text section
    jal     r6
    switch  xdata            ; use now data section
value:
    dc.b    1,2,3            ; fills data section
```

The assembler allows up to **255** different sections. A section name is limited to **15** characters. If a section name is too long, it is simply truncated without any error message.

```
xref          var
```

Includes

The **include** directive specifies a file to be included and assembled in place of the **include** directive. The file name is written between double quotes, and may be any character string describing a file on the host system. If the file cannot be found using the given name, it is searched from all the include paths defined by the **-i** options on the command line, and from the paths defined by the environment symbol **CXLIB**, if such a symbol has been defined before the assembler invocation. This symbol may contain several paths separated by the usual path separator of the host operating system (‘;’ for MSDOS and ‘:’ for UNIX).

The **-h** option can specify a file to be “included”. The file specified will be included as if the program had an **include** directive at its very top. The specified file will be included before **any** source file specified on the command line.

Branch Optimization

Branch instructions are by default automatically optimized to produce the shortest code possible. This behaviour may be disabled by the **-b** option. This optimization operates on conditional branches, on jumps and jumps to subroutine.

A *conditional* branch offset is limited to the range [-512,511]. If such an instruction cannot be encoded properly, the assembler will replace it by a sequence containing an inverted branch to the next location followed immediately by a jump to the original target address. The assembler keep track of the last replacement for each label, so if a long branch has already been expanded for the same label at a location close enough from the current instruction, the target address of the short branch will be changed only to branch on the already existing jump instruction to the specified label.

```

    beq    farlabel  becomes      bne    *+5
                                bra     farlabel

```

The sequence *ldw + jal* instructions can be replaced by a shorter sequence for the second silicon version, if the target function is in the range of a *bra* instruction. In order to allow the assembler to automatically choose the best coding, a special syntax is provided, specifying both register and target address:

```

    jal    r6, _func

```

will be expanded as:

```

    tfr    r6, pc          or      ldw    r6, #_func
    bra    _func           jal     r6

```

depending on the location of the *_func* symbol.

C Style Directives

The assembler also supports C style directives matching the preprocessor directives of a C compiler. The following directives list shows the equivalence with the standard directives:

C Style	Assembler Style
#include "file"	include "file"
#define label expression	label: equ expression
#define label	label: equ 1
#if expression	if expression
#ifdef label	ifdef label
#ifndef label	ifndef label
#else	else
#endif	endif
#error "message"	fail "message"

NOTE

The #define directive does not implement all the text replacement features provided by a C compiler. It can be used only to define a symbol equal to a numerical value.

Assembler Directives

This section consists of quick reference descriptions for each of the *cax-gate* assembler directives.

align

Description

Align the next instruction on a given boundary

Syntax

```
align <expression>, [<fill_value>]
```

Function

The **align** directive forces the next instruction to start on a specific boundary. The **align** directive is followed by a constant expression which must be positive. The next instruction will start at the next address which is a multiple of the specified value. If bytes are added in the section, they are set to the value of the filling byte defined by the **-f** option. If **<fill_value>**, is specified, it will be used locally as the filling byte, instead of the one specified by the **-f** option.

Example

```
align 3      ; next address is multiple of 3
ds.b 1
```

See Also

even

base

Description

Define the default base for numerical constants

Syntax

```
base <expression>
```

Function

The **base** directive sets the default base for numerical constants beginning with a digit. The **base** directive is followed by a constant expression which value must be one of **2**, **8**, **10** or **16**. The decimal base is used by default. When another base is selected, it is no more possible to enter decimal constants.

Example

```
base    8                ; select octal base
ldl     r2,#377          ; load $FF
```

bsct

Description

Switch to the predefined **.bsct** section.

Syntax

bsct

Function

The **bsct** directive switches input to a section named **.bsct**, also known as the **zero page** section. The assembler will automatically select the direct addressing mode when referencing an object defined in the *.bsct* section.

Example

```
        bsct
c_reg:
        ds.b    1
```

Notes

This directive is not relevant for the XGATE assembler.

See Also

section, switch

clist

Description

Turn listing of conditionally excluded code on or off.

Syntax

```
clist [on|off]
```

Function

The **clist** directive controls the output in the listing file of conditionally excluded code. It is effective if and only if listings are requested; it is ignored otherwise.

The parts of the program to be listed are the program lines which are not assembled as a consequence of **if**, **else** and **endif** directives.

See Also

if, else, endif

dc

Description

Allocate constant(s)

Syntax

`dc [.size] <expression>[,<expression>...]`

Function

The **dc** directive allocates and initializes storage for constants. If *<expression>* is a string constant, one byte is allocated for each character of the string. Initialization can be specified for each item by giving a series of values separated by commas or by using a repeat count.

The **dc** and **dc.b** directives will allocate one byte per *<expression>*.

The **dc.w** directive will allocate one word per *<expression>*.

The **dc.l** directive will allocate one long word per *<expression>*.

Example

```
digit:dc.b    10,'0123456789'  
          dc.w    digit
```

Note

For compatibility with previous assemblers, the directive **fcb** is alias to **dc.b**, and the directive **fdb** is alias to **dc.w**.

dcb

Description

Allocate constant block

Syntax

```
dcb.<size> <count>,<value>
```

Function

The **dcb** directive allocates a memory block and initializes storage for constants. The size area is the number of the specified value *<count>* of *<size>*. The memory area can be initialized with the *<value>* specified.

The **dcb** and **dcb.b** directives will allocate one **byte** per *<count>*.

The **dcb.w** directive will allocate one **word** per *<count>*.

The **dcb.l** directive will allocate one **long word** per *<count>*.

Example

```
digit: dcb.b 10,5    ; allocate 10 bytes,  
                    ; all initialized to 5
```

dlist

Description

Turn listing of debug directives on or off.

Syntax

```
dlist [on|off]
```

Function

The **dlist** directive controls the visibility of any debug directives in the listing. It is effective if and only if listings are requested; it is ignored otherwise.

ds

Description

Allocate variable(s)

Syntax

```
ds[.size] <space>
```

Function

The **ds** directive allocates storage space for variables. *<space>* must be an absolute expression. Bytes created are set to the value of the filling byte defined by the **-f** option.

The **ds** and **ds.b** directives will allocate *<space>* bytes.

The **ds.w** directive will allocate *<space>* words.

The **ds.l** directive will allocate *<space>* long words.

Example

```
ptlec:      ds.b    2
ptecr:      ds.b    2
chrbuf:     ds.w    128
```

Note

For compatibility with previous assemblers, the directive **rmb** is alias to **ds.b**.

else

Description

Conditional assembly

Syntax

```
if <expression>
instructions
else
instructions
endif
```

Function

The **else** directive follows an **if** directive to define an alternative conditional sequence. It reverts the condition status for the following instructions up to the next matching **endif** directive. An **else** directive applies to the closest previous **if** directive.

Example

```
if    offset != 1           ; if offset too large
    addptr offset          ; call a macro
else
    add    r2,#1           ; increment R2 register
endif
```

Note

The **else** and **elsec** directives are equivalent and may be used without distinction. They are provided for compatibility with previous assemblers.

See Also

if, endif, clist

elsec

Description

Conditional assembly

Syntax

```
if <expression>
instructions
elsec
instructions
endc
```

Function

The **elsec** directive follows an **if** directive to define an alternative conditional sequence. It reverts the condition status for the following instructions up to the next matching **endc** directive. An **elsec** directive applies to the closest previous **if** directive.

Example

```
ifge offset-255          ; if offset too large
    addptr offset        ; call a macro
elsec                    ; otherwise
    addl    r2,#offset   ; increment R2 register
endc
```

Note

The **elsec** and **else** directives are equivalent and may be used without distinction. They are provided for compatibility with previous assemblers.

See Also

if, endc, clist, else

end

Description

Stop the assembly

Syntax

```
end
```

Function

The **end** directive stops the assembly process. Any statements following it are ignored. If the **end** directive is encountered in an included file, it will stop the assembly process for the included file only.

endc

Description

End conditional assembly

Syntax

```
if<cc> <expression>
instructions
endc
```

Function

The **endc** directive closes an **if<cc>** or **elsec** conditional directive. The conditional status reverts to the one existing before entering the **if<cc>** directives. The **endc** directive applies to the closest previous **if<cc>** or **elsec** directive.

Example

```
ifge offset-255          ; if offset too large
    addptr offset        ; call a macro
elsec                    ; otherwise
    addl    r2,#offset    ; increment R2 register
endc
```

Note

The **endc** and **endif** directives are equivalent and may be used without distinction. They are provided for compatibility with previous assemblers.

See Also

if<cc>, elsec, clist, end

endif

Description

End conditional assembly

Syntax

```
if <expression>
instructions
endif
```

Function

The **endif** directive closes an **if** or **else** conditional directive. The conditional status reverts to the one existing before entering the **if** directive. The **endif** directive applies to the closest previous **if** or **else** directive.

Example

```
if    offset != 1           ; if offset too large
    addptr offset          ; call a macro
else  ; otherwise
    add    r2,#1           ; increment R2 register
endif
```

Note

The **endif** and **endc** directives are equivalent and may be used without distinction. They are provided for compatibility with previous assemblers.

See Also

if, else, clist

endm

Description

End macro definition

Syntax

```
label: macro  
    <macro_body>  
endm
```

Function

The **endm** directive is used to terminate macro definitions.

Example

```
; define a macro that places the length of  
; a string in a byte prior to the string
```

```
ltext:macro  
    ds.b    \@2 - \@1  
    \@1:  
    ds.b    \1  
    \@2:  
endm
```

See Also

mexit, macro

endr

Description

End repeat section

Syntax

```
repeat  
<macro_body>  
endr
```

Function

The **endr** directive is used to terminate repeat sections.

Example

```
; shift a value n times  
asln: macro  
    repeat \1  
        lsl    r2,#1  
    endr  
endm  
  
; use of above macro  
asln 10          ;shift 10 times
```

See Also

repeat

equ

Description

Give a permanent value to a symbol

Syntax

```
label: equ <expression>
```

Function

The **equ** directive is used to associate a permanent value to a symbol (label). Symbols declared with the **equ** directive may not subsequently have their value altered otherwise the **set** directive should be used. *<expression>* must be either a constant expression, or a relocatable expression involving a symbol declared in the same section as the current one.

Example

```
false:equ 0           ; initialize these values
true: equ 1
tablen:equ tabfin - tabsta ;compute table length
nul:  equ $0           ; define strings for ascii characters
soh:  equ $1
stx:  equ $2
etx:  equ $3
eot:  equ $4
enq:  equ $5
```

See Also

lit, set

even

Description

Assemble next byte at the next even address relative to the start of a section.

Syntax

```
even [<fill_value>]
```

Function

The **even** directive forces the next assembled byte to the next even address. If a byte is added to the section, it is set to the value of the filling byte defined by the **-f** option. If **<fill_value>**, is specified, it will be used locally as the filling byte, instead of the one specified by the **-f** option.

Example

```
vowtab: dc.b 'aeiou'
        even                ; ensure aligned at even address
tentab: dc.w 1, 10, 100, 1000
```


fail

Description

Generate error message.

Syntax

```
fail "string"
```

Function

The **fail** directive outputs “*string*” as an error message. No output file is produced as this directive creates an assembly error. *fail* is generally used with conditional directives.

Example

```
Max:  equ    512
      ifge   value - Max
      fail   "Value too large"
```

if

Description

Conditional assembly

Syntax

if <expression> instructions endif	or	if <expression> instructions else instructions endif
--	----	---

Function

The **if**, **else** and **endif** directives allow conditional assembly. The **if** directive is followed by a constant expression. If the result of the expression is **not** zero, the following instructions are assembled up to the next matching **endif** or **else** directive; otherwise, the following instructions up to the next matching **endif** or **else** directive are skipped.

If the **if** statement ends with an **else** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endif**. So, if the **if** expression was **not** zero, the instructions between **else** and **endif** are skipped; otherwise, the instructions between **else** and **endif** are assembled. An **else** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

Example

```
if      offset != 1           ; if offset too large
    addptr offset           ; call a macro
else    ; otherwise
    add    r2,#1            ; increment R2 register
endif
```

See Also

else, endif, clist

ifc

Description

Conditional assembly

Syntax

ifc <string1>,<string2>	or ifc <string1>,<string2>
instructions	instructions
endc	elsec
	instructions
	endc

Function

The **ifc**, **else** and **endc** directives allow conditional assembly. The **ifc** directive is followed by a constant expression. If <string1> and <string2> are equals, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped.

If the **ifc** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endc**. So, if the **ifc** expression was **not** zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

Example

```

ifc    "hello", \2           ; if "hello" equals argument
    ldl    r2,#45           ; load 45
elsec
    ldl    r2,#0           ; otherwise...
endc

```

See Also

elsec, endc, clist

ifdef

Description

Conditional assembly

Syntax

ifdef <label> instructions endc	or	ifdef <label> instructions elsec instructions endc
--	-----------	---

Function

The **ifdef**, **elsec** and **endc** directives allow conditional assembly. The **ifdef** directive is followed by a label <label>. If <label> is defined, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped. <label> must be first defined. It cannot be a forward reference.

If the **ifdef** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endif**. So, if the **ifdef** expression was **not** zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

Example

```
ifdef offset1           ; if offset1 is defined
    addptr offset1      ; call a macro
elsec                   ; otherwise
    addptr offset2      ; call a macro
endif
```

See Also

ifndef, elsec, endc, clist

ifeq

Description

Conditional assembly

Syntax

ifeq <expression> instructions endc	or	ifeq <expression> instructions elsec instructions endc
--	----	---

Function

The **ifeq**, **elsec** and **endc** directives allow conditional assembly. The **ifeq** directive is followed by a constant expression. If the result of the expression is **equal** to zero, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped.

If the **ifeq** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endc**. So, if the **ifeq** expression is **equal** to zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

Example

```

ifeq  offset                ; if offset nul
      cmp    r2,#0          ; just test it
elsec                ; otherwise
      add    r2,#offset      ; add to accu
endc
```

See Also

elsec, endc, clist

ifge

Description

Conditional assembly

Syntax

ifge <expression> instructions endc	or	ifge <expression> instructions elsec instructions endc
--	----	---

Function

The **ifge**, **elsec** and **endc** directives allow conditional assembly. The **ifge** directive is followed by a constant expression. If the result of the expression is **greater or equal** to zero, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped.

If the **ifge** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endc**. So, if the **ifge** expression is **greater or equal** to zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

Example

```
ifge  offset-255           ; if offset too large
      addptr offset        ; call a macro
elsec                               ; otherwise
      addl   r2,#offset    ; increment R2 register
endc
```

See Also

elsec, endc, clist

ifgt

Description

Conditional assembly

Syntax

ifgt <expression> instructions endc	or	ifgt <expression> instructions elsec instructions endc
--	----	---

Function

The **ifgt**, **elsec** and **endc** directives allow conditional assembly. The **ifgt** directive is followed by a constant expression. If the result of the expression is **greater than** zero, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped.

If the **ifgt** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endc**. So, if the **ifgt** expression was **greater** than zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

Example

```

ifgt  offset-255           ; if offset too large
    addptr offset         ; call a macro
elsec                               ; otherwise
    addl  r2,#offset      ; increment R2 register
endc

```

See Also

elsec, endc, clist

ifl

Description

Conditional assembly

Syntax

ifl <expression> instructions endc	or	ifl <expression> instructions elsec instructions endc
---	----	--

Function

The **ifl**, **elsec** and **endc** directives allow conditional assembly. The **ifl** directive is followed by a constant expression. If the result of the expression is **less or equal** to zero, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped.

If the **ifl** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endc**. So, if the **ifl** expression was **less or equal** to zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

Example

```
ifl   offset-255           ; if offset small enough
      addl   r2,#offset    ; increment R2 register
elsec           ; otherwise
      addptr offset        ; call a macro
endc
```

See Also

elsec, endc, clist

iflt

Description

Conditional assembly

Syntax

iflt <expression> instructions endc	or	iflt <expression> instructions elsec instructions endc
---	----	---

Function

The **iflt**, **elsec** and **endc** directives allow conditional assembly. The **iflt** directive is followed by a constant expression. If the result of the expression is **less than** zero, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped.

If the **iflt** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endc**. So, if the **iflt** expression was **less than** zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

Example

```

iflt  offset-255           ; if offset small enough
    addl  r2,#offset       ; increment R2 register
elsec                               ; otherwise
    addptr offset         ; call a macro
endc

```

See Also

elsec, endc, clist

ifndef

Description

Conditional assembly

Syntax

ifndef <label> instructions endc	or	ifndef <label> instructions elsec instructions endc
---	-----------	--

Function

The **ifndef**, **else** and **endc** directives allow conditional assembly. The **ifndef** directive is followed by a label <label>. If <label> is not defined, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped. <label> must be first defined. It cannot be a forward reference.

If the **ifndef** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endif**. So, if the **ifndef** expression was **not** zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

Example

```
ifndef offset1           ; if offset1 is not defined
    addptr offset2       ; call a macro
elsec                    ; otherwise
    addptr offset1       ; call a macro
endif
```

See Also

ifdef, elsec, endc, clist

ifne

Description

Conditional assembly

Syntax

ifne <expression> instructions endc	or	ifne <expression> instructions elsec instructions endc
---	----	---

Function

The **ifne**, **elsec** and **endc** directives allow conditional assembly. The **ifne** directive is followed by a constant expression. If the result of the expression is **not equal** to zero, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped.

If the **ifne** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endc**. So, if the **ifne** expression was **not equal** to zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

Example

```

ifne  offset           ; if offset not nul
      add    r2,#offset ; add to register
elsec           ; otherwise
      cmp    r2,#0      ; just test it
endc

```

See Also

elsec, endc, clist

ifnc

Description

Conditional assembly

Syntax

ifnc <string1>,string2>	or ifnc <string1><string2>
instructions	instructions
endc	elsec
	instructions
	endc

Function

The **ifnc**, **elsec** and **endc** directives allow conditional assembly. The **ifnc** directive is followed by a constant expression. If <string1> and <string2> are different, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped.

If the **ifnc** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endc**. So, if the **ifnc** expression was **not** zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

Example

```
ifnc  "hello", \2
      addptr offset      ; call a macro
else
      add    r2,#1       ; increment R2 register
endif
```

See Also

elsec, endc, clist

include

Description

Include text from another text file

Syntax

```
include "filename"
```

Function

The **include** directive causes the assembler to switch its input to the specified *filename* until end of file is reached, at which point the assembler resumes input from the line following the **include** directive in the current file. The directive is followed by a string which gives the name of the file to be included. This string must match exactly the name and extension of the file to be included; the host system convention for uppercase/lowercase characters should be respected.

Example

```
include "datstr" ; use data structure library
include "bldstd" ; use current build standard
include "matmac" ; use maths macros
include "ports82" ; use ports definition
```

list

Description

Turn on listing during assembly.

Syntax

```
list
```

Function

The **list** directive controls the parts of the program which will be written to the listing file. It is effective if and only if listings are requested; it is ignored otherwise.

Example

```
list    ; expand source code until end or nolist
dc.b 1,2,4,8,16
end
```

See Also

nolist

lit

Description

Give a text equivalent to a symbol

Syntax

```
label: lit "string"
```

Function

The **lit** directive is used to associate a text string to a symbol (label). This symbol is replaced by the string content when parsed in any assembler instruction or directive.

Example

```
nbr:  lit    "#5"
      ldl    r2,nbr      ; expand as 'ldl r2,#5'
```

See Also

equ, set

local

Description

Create a new local block

Syntax

<code>local</code>

Function

The **local** directive is used to create a new local block. When the *local* directive is used, all temporary labels defined before the local directive will be undefined after the local label. New local labels can then be defined in the new local block. Local labels can only be referenced within their own local block. A local label block is the area between two standard labels or local directives or a combination of the two.

Example

```
var:  ds.b    1
var2: ds.b    1
function1:
10$:  cmp     r2,r1
      beq     10$
      sub     r2,r2,r1
local
10$:  cmp     r1,r2
      beq     10$
      sub     r1,r1,r2
      rts
```


macro

Description

Define a macro

Syntax

```
label: macro    <argument_list>
    <macro_body>
endm
```

Function

The **macro** directive is used to define a macro. The name may be any previously unused name, a name already used as a macro, or an instruction mnemonic for the microprocessor.

Macros are expanded when the name of a previously defined macro is encountered. Operands, where given, follow the name and are separated from each other by commas.

The *<argument_list>* is optional and, if specified, is declaring each argument by name. Each argument name is prefixed by a `\` character, and separated from any other name by a comma. An argument name is an identifier which may contain `.` and `_` characters.

The *<macro_body>* consists of a sequence of instructions not including the directives **macro** or **endm**. It may contain macro variables which will be replaced, when the macro is expanded, by the corresponding operands following the macro invocation. These macro variables take the form `\1` to `\9` to denote the first to ninth operand respectively and `\A` to `\Z` to denote the tenth to 35th operand respectively, if the macro has been defined without any *<argument_list>*. Otherwise, macro variables are denoted by their name prefixed by a `\` character. The macro variable name can also be enclosed by parenthesis to avoid unwanted concatenation with the remaining text. In addition, the macro variable `\#` contains the number of actual operands for a macro invocation.

The special parameter `*` is expanded to the full list of passed arguments separated by commas.

The special parameter `\0` corresponds to an extension `<ext>` which may follow the macro name, separated by the period character `'.'`. For more information, see “[Macro Instructions](#)” on page 175.

A macro expansion may be terminated early by using the **mexit** directive which, when encountered, acts as if the end of the macro has been reached.

The sequence `'\@'` may be inserted in a label in order to allow a unique name expansion. The sequence `'\@'` will be replaced by a unique number.

A macro can not be defined within another macro.

Example

```
; define a macro that places the length of a string
; in a byte in front of the string using numbered syntax
;
ltext:macro
    dc.b    \@2-\@1
\@1:
    dc.b    \1      ; text given as first operand
\@2:
    endm

; define a macro that places the length of a string
; in a byte in front of the string using named syntax
;
ltext:macro \string
    dc.b    \@2-\@1
\@1:
    dc.b    \string ; text given as first operand
\@2:
    endm
```

See Also

endm, mexit

messg

Description

Send a message out to STDOUT

Syntax

```
messg "<text>"  
messg '<text>'
```

Function

The **messg** directive is used to send a message out to the host system's standard output (STDOUT).

Example

```
messg "Test code for debug"  
ldl    r2, #2  
stw    r2, (r1, #4)
```

See Also

title

mexit

Description

Terminate a macro definition

Syntax

```
mexit
```

Function

The **mexit** directive is used to exit from a macro definition before the **endm** directive is reached. *mexit* is usually placed after a conditional assembly directive.

Example

```
ctrace:macro
    if tflag == 0
    mexit
    endif
    jal    \1
endm
```

See Also

endm, macro

mlist

Description

Turn on or off listing of macro expansion.

Syntax

```
mlist [on|off]
```

Function

The **mlist** directive controls the parts of the program which will be written to the listing file produced by a macro expansion. It is effective if and only if listings are requested; it is ignored otherwise.

The parts of the program to be listed are the lines which are assembled in a macro expansion.

See Also

macro

nolist

Description

Turn off listing.

Syntax

```
nolist
```

Function

The **nolist** directive controls the parts of the program which will be **not** written to the listing file until an **end** or a **list** directive is encountered. It is effective if and only if listings are requested; it is ignored otherwise.

See Also

list

Note

For compatibility with previous assemblers, the directive **nol** is alias to **nolist**.

nopage

Description

Disable pagination in the listing file

Syntax

nopage

Function

The **nopage** directive stops the pagination mechanism in the listing output. It is ignored if no listing has been required.

Example

```
xref    mult, div
nopage
ds.b    charin, charout
ds.w    a, b, sum
```

See Also

plen, title

offset

Description

Creates absolute symbols

Syntax

```
offset <expression>
```

Function

The **offset** directive starts an absolute section which will only be used to define symbols, and not to produce any code or data. This section starts at the address specified by *<expression>*, and remains active while no directive or instructions producing code or data is entered. This absolute section is then destroyed and the current section is restored to the one which was active when the *offset* directive has been entered. All the labels defined in this section become absolute symbols.

<expression> must be a valid absolute expression. It must not contain any forward or external references.

Example

```
        offset 0
next:
        ds.b    2
buffer:
        ds.b    80

        switch .text
size:
        ldw     r2,(r1,#next) ; ends the offset section
```


org

Description

Sets the location counter to an offset from the beginning of a section.

Syntax

```
org <expression>
```

Function

<expression> must be a valid absolute expression. It must not contain any forward or external references.

For an absolute section, the first **org** before any code or data defines the starting address.

An *org* directive cannot define an address smaller than the location counter of the current section.

Any gap created by an *org* directive is filled with the byte defined by the **-f** option.

page

Description

Start a new page in the listing file

Syntax

page

Function

The **page** directive causes a formfeed to be inserted in the listing output if pagination is enabled by either a **title** directive or the **-ft** option.

Example

```
xref    mult, div
page
ds.b     charin, charout
ds.w     a, b, sum
```

See Also

plen, title

plen

Description

Specify the number of lines per pages in the listing file

Syntax

```
plen <page_length>
```

Function

The **plen** directive causes *<page_length>* lines to be output per page in the listing output if pagination is enabled by either a **title** directive or the **-ft** option. If the number of lines already output on the current page is less than *<page_length>*, then the new page length becomes effective with *<page_length>*. If the number of lines already output on the current page is greater than or equal to *<page_length>*, a new page will be started and the new page length is set to *<page_length>*.

Example

```
plen    58
```

See Also

page, title

repeat

Description

Repeat a list of lines a number of times

Syntax

```
repeat <expression>
    repeat_body
endr
```

Function

The **repeat** directive is used to cause the assembler to repeat the following list of source line up to the next **endr** directive. The number of times the source lines will be repeated is specified by the expression operand. The **repeat** directive is equivalent to a macro definition followed by the same number of calls on that macro.

Example

```
; shift a value n times
asln: macro
    repeat \1
    lsl    r2,#1
    endr
endm

; use of above macro
asln    5
```

See Also

endr, repeatl, rexit

repeatl

Description

Repeat a list of lines a number of times

Syntax

```
repeatl <arguments>
    repeat_body
endr
```

Function

The **repeatl** directive is used to cause the assembler to repeat the following list of source line up to the next **endr** directive. The number of times the source lines will be repeated is specified by the number of arguments, separated with commas (with a maximum of 36 arguments) and executed each time with the value of an argument. The **repeatl** directive is equivalent to a macro definition followed by the same number of calls on that macro with each time a different argument. The repeat argument is denoted **\1** unless the argument list is starting by a name prefixed by a **** character. In such a case, the repeat argument is specified by its name prefixed by a **** character.

A **repeatl** directive may be terminated early by using the **rexit** directive which, when encountered, acts as if the end of the **repeatl** has been reached.

Example

```
; test a value using the numbered syntax
repeatl      1,2,3
    add      r2,#\1          ; add to register
endr
end

or

; test a value using the named syntax
repeatl      \count,1,2,3
    add      r2,#\count      ; add to register
endr
end
```

will both produce:

```
2          ; test a value
9 0000 ab01 add    r2,#1      ; add to register
9 0002 ab02 add    r2,#2      ; add to register
9 0004 ab03 add    r2,#3      ; add to register
10          end
```

See Also

endr, repeat, rexit

restore

Description

Restore saved section

Syntax

```
restore
```

Function

The **restore** directive is used to restore the last saved section. This is equivalent to a switch to the saved section.

Example

```
switch.bss
var:  ds.b 1
var2: ds.b 1
      save
      switch .text

function1:
10$:  cmp    r2,r1
      beq    10$
      sub    r2,r2,r1

function2:
10$:  cmp    r1,r2
      bne    10$
      rts
      restore
var3:  ds.b 1
var4:  ds.b 1

      switch .text

      ldw    r1,r2

end
```

See Also

save, section

rexit

Description

Terminate a repeat definition

Syntax

```
rexit
```

Function

The **rexit** directive is used to exit from a **repeat** definition before the **endr** directive is reached. *rexit* is usually placed after a conditional assembly directive.

Example

```
        ; shift a value n times
asln: macro
    repeat \1
    if \1 == 0
        rexit
    endif
    lsl    r2,#1
    endr
endm

        ; use of above macro
asln    5
```

See Also

endr, repeat, repeatl

save

Description

Save section

Syntax

save

Function

The **save** directive is used to save the current section so it may be restored later in the source file.

Example

```
switch .bss
var:  ds.b 1
var2: ds.b 1
      save
      switch .text

function1:
10$:  cmp    r2,r1
      beq    10$
      sub    r2,r2,r1

function2:
10$:  cmp    r1,r2
      bne    10$
      rts
      restore
var3:  ds.b 1
var4:  ds.b 1

      switch .text

      ldw    r1,r2

end
```

See Also

restore, section

section

Description

Define a new section

Syntax

```
<section_name>: section [<attributes>]
```

Function

The **section** directive defines a new section, and indicates that the following program is to be assembled into a section named *<section_name>*. The *section* directive cannot be used to redefine an already existing section. If no name and no attributes are specified to the section, the default is to define the section as a *text* section with its same attributes. It is possible to associate *<attributes>* to the new section. An attribute is either the name of an existing section or an attribute keyword. Attributes may be added if prefixed by a '+' character or not prefixed, or deleted if prefixed by a '-' character. Several attributes may be specified separated by commas. Attribute keywords are:

abs	absolute section
bss	bss style section (no data)
hilo	values are stored in descending order of significance
even	enforce even starting address and size
zpage	enforce 8 bit relocation
long	enforce 32 bit relocation
bit	bit section

Example

```
CODE: section      .text           ; section of text
lab1: ds.b         5
DATA: section      .data          ; section of data
lab2: ds.b         6
      switch
lab3: ds.b         7
      switch
lab4: ds.b         8
      DATA
```

This will place **lab1** and then **lab3** into consecutive locations in section CODE and **lab2** and **lab4** in consecutive locations in section DATA.

```
.frame: section      .bsct,even
```

The *.frame* section is declared with same attributes than the *.bsct* section and with the *even* attribute.

```
.bit: section      +zpage,+even,-hilo
```

The *.bit* section is declared using 8 bit relocation, with an even alignment and storing data with an ascending order of significance.

When the **-m** option is used, the *section* directive also accepts a number as operand. In that case, a labelled directive is considered as a section definition, and an unlabelled directive is considered as a section opening (*switch*).

```
.rom: section1      ; define section 1
      nop
.ram: section2      ; define section 2
      dc.b 1
      section1      ; switch back to section 1
      nop
```

It is still possible to add attributes after the section number of a section definition line, separated by a comma.

See Also

switch, bsct

set

Description

Give a resetable value to a symbol

Syntax

```
label: set <expression>
```

Function

The **set** directive allows a value to be associated with a symbol. Symbols declared with **set** may be altered by a subsequent **set**. The **equ** directive should be used for symbols that will have a constant value. *<expression>* must be fully defined at the time the **equ** directive is assembled.

Example

```
OFST: set    10
```

See Also

equ, lit

spc

Description

Insert a number of blank lines before the next statement in the listing file.

Syntax

```
spc <num_lines>
```

Function

The **spc** directive causes <num_lines> blank lines to be inserted in the listing output before the next statement.

Example

```
spc      5
title    "new file"
```

If listing is requested, 5 blank lines will be inserted, then the title will be output.

See Also

title

switch

Description

Place code into a section.

Syntax

```
switch <section_name>
```

Function

The **switch** directive switches output to the section defined with the **section** directive. *<section_name>* is the name of the target section, and has to be already defined. All code and data following the *switch* directive up to the next *section*, *switch*, *bsct* or *end* directive are placed in the section *<section_name>*.

Example

```
switch .bss
buffer:ds.b 512
xdef      buffer
```

This will place **buffer** into the *.bss* section.

See Also

section, *bsct*

tabs

Description

Specify the number of spaces for a tab character in the listing file

Syntax

```
tabs <tab_size>
```

Function

The **tabs** directive sets the number of spaces to be substituted to the tab character in the listing output. The minimum value of *<tab_size>* is 0 and the maximum value is 128.

Example

```
tabs    6
```

title

Description

Define default header

Syntax

```
title "name"
```

Function

The **title** directive is used to enable the listing pagination and to set the default page header used when a new page is written to the listing output.

Example

```
title "My Application"
```

See Also

messg, page, plen

Note

For compatibility with previous assemblers, the directive **ttl** is alias to **title**.

xdef

Description

Declare a variable to be visible

Syntax

```
xdef identifier[,identifier...]
```

Function

Visibility of symbols between modules is controlled by the **xdef** and **xref** directives. A symbol may only be declared as *xdef* in one module. A symbol may be declared both *xdef* and *xref* in the same module, to allow for usage of common headers.

Example

```
xdef    sqrt    ; allow sqrt to be called
                ; from another module
sqrt:    ; routine to return a square root
                ; of a number >= zero
```

See Also

xref

xref

Description

Declare symbol as being defined elsewhere

Syntax

```
xref[.b] identifier[,identifier...]
```

Function

Visibility of symbols between modules is controlled by the **xref** and **xdef** directives. Symbols which are defined in other modules must be declared as *xref*. A symbol may be declared both *xdef* and *xref* in the same module, to allow for usage of common headers.

The directive *xref.b* declares external symbols located in the *.bsct* section.

Example

```
xref    otherprog
xref.b  zpage      ; is in .bsct section
```

See Also

xdef

CHAPTER 6

Using The Linker

As the XGATE compiler is an add-on to the S12X compiler, the linker is not provided. Please report to **chapter 6** of the COSMIC “*C Cross Compiler Users’ Guide for S12X*”.

This chapter explains only the specific needs required by the XGATE features. It also provides an example linker command line that shows you how to perform some useful operations. This chapter includes the following sections:

- Linking XGATE Objects
- Linking Library Objects

Linking XGATE Objects

XGATE code and data are produced in specific sections which have no default behaviour. Each section must be explicitly located. The XGATE processor accesses memory with a different coding than the S12X. The **-b** and **-o** options will be used whenever necessary to distinguish addresses in the XGATE range from addresses in the S12X range. Some areas will be shared by both processor (mainly data objects), the **-pr** option (physical relocation) will be used to tell the linker to relocate addresses using the **-b** value instead of the **-o** one.

The XGATE code is designed to be executed from the S12X ram space. It has to be created as initialized segments in order to be handled by the automatic data initialization feature of the *crtss.s* startup file which must be linked in the S12X part of the application.

The linker command file should contain such directives before loading XGATE object files:

```
1 # XGATE definitions
2 +seg .xtext -b0xfb000 -m0x2000 -pr -n.xtext -id# code
3 +seg .xconst -a.xtext -pr -n.xconst -id# xgate constants
4 +seg .xdata -b0xfd000 -o0x1000 -m0x1000 -pr -n.xdata -id
5 +seg .xbss -a.xdata -pr -n.xbss
6 +def __extext=pend(.xconst) # end of xgate code
7 +def __sxdata=pstart(.xdata) # start of shared data
8 +def __exdata=pend(.xbss) # end of shared data
9 # enter object files for XGATE here
10 xvector.o # xgate vectors
11 +pri
12 libi.xgt # xgate C library (if needed)
13 +new
14 libm.xgt # xgate library
15 +spc .xbss=64 # xgate stack
16 +def __xstack=@.xbss # if needed
```

The code and constant segments, on lines 2 and 3, are declared with only **-b** option because they do not need to be accessed by the S12X code. The **-id** option must be specified in order to allow the initialization process.

The data segments, on lines 4 and 5, are using the both **-b** and **-o** options, matching respectively the XGATE address value and the S12X address value. The **-id** option is specified only on the initialized data segment.

The symbol definitions on lines 6 to 8 can be used to initialize the ram protection registers allowing to mark which areas are accessed only by XGATE, or only by S12X or by both.

The integer library, if necessary, must be linked in a private region using the **+pri** and **+new** directives in order to avoid any conflict with the S12X library functions. The machine library is using non conflicting names and does not need such a precaution.

The symbol `__xstack` created on line 16 is used to indicate where the XGATE stack is starting. The stack space is created here by the space reservation on line 15, but there are other possibilities for such a symbol to be created (absolute value, C object address,...).

Linking Library Objects

The linker will selectively include modules from a library when outstanding references to member functions are encountered. The library file must be placed **after** all objects that may call its modules to avoid unresolved references. The standard ANSI libraries are provided in two versions to provide the level of support that your application needs. This can save a significant amount of code space and execution time when full ANSI single precision floating point support is not needed. The first letter after “*lib*” in each library file denotes the library type (**f** for single precision, and **i** for integer). See below.

libf.xgt Floating Point Library. This library is used for applications where only single precision floating point support is needed. Link this library **before** the other libraries when **only** single precision floats are used.

libi.xgt Integer only Library. This library is designed for applications where **no** floating point is used. Floats can still be used for arithmetic but not with the standard library. Link

this library *before* the other libraries when only integer libraries are needed.

NOTE

The XGATE integer library does not implement all the functions available in the S12X integer library.

CHAPTER 7

Debugging Support

As the XGATE compiler is an add-on to the S12X compiler, the debugging support utilities are not provided. Please report to **chapter 7** of the COSMIC “*C Cross Compiler Users’ Guide for S12X*”.

CHAPTER 8

Programming Support

As the XGATE compiler is an add-on to the S12X compiler, the programming support utilities are not provided. Please report to **chapter 8** of the COSMIC “*C Cross Compiler Users' Guide for S12X*”.

Compiler Error Messages

This appendix lists the error messages that the compiler may generate in response to errors in your program, or in response to problems in your host system environment, such as inadequate space for temporary intermediate files that the compiler creates.

The first pass of the compiler generally produces all user diagnostics. This pass deals with # control lines and lexical analysis, and then with everything else having to do with semantics. Only machine-dependent extensions are diagnosed in the code generator pass. If a pass produces diagnostics, later passes will not be run.

Any compiler message containing an exclamation mark **!** or the word **'PANIC'** indicates that the compiler has detected an inconsistent internal state. Such occurrences are uncommon and should be reported to the maintainers.

- [Parser \(cpxgate\) Error Messages](#)
- [Code Generator \(cgxgate\) Error Messages](#)
- [Assembler \(caxgate\) Error Messages](#)
- [Linker \(clnk\) Error Messages](#)

Parser (cpxgate) Error Messages

<name> not a member - field name not recognized for this struct/
union

<name> not an argument - a declaration has been specified for an
argument not specified as a function parameter

<name> undefined - a function or a variable is never defined

FlexLM <message> - an error is detected by the license manager

_asm string too long - the string constant passed to *_asm* is larger than
255 characters

ambiguous space modifier - a space modifier attempts to redefine an
already specified modifier

array size unknown - the *sizeof* operator has been applied to an array
of unknown size

bad # argument in macro <name> - the argument of a # operator in a
#define macro is not a parameter

bad # directive: <name> - an unknown *#directive* has been specified

bad # syntax - # is not followed by an identifier

bad ## argument in macro <name> - an argument of a ## operator in
a *#define* macro is missing

bad #asm directive - a *#asm* directive is not entered at a valid declara-
tion or instruction boundary

bad #define syntax - a *#define* is not followed by an identifier

bad #elif expression - a *#elif* is not followed by a constant expression

bad #else - a *#else* occurs without a previous *#if*, *#ifdef*, *#ifndef* or *#elif*

bad #endasm directive - a *#endasm* directive is not closing a previous
#asm directive

bad #endif - a *#endif* occurs without a previous *#if*, *#ifdef*, *#ifndef*, *#elif* or *#else*

bad #if expression - the expression part of a *#if* is not a constant expression

bad #ifdef syntax - extra characters are found after the symbol name

bad #ifndef syntax - extra characters are found after the symbol name

bad #include syntax - extra characters are found after the file name

bad #pragma section directive - syntax for the *#pragma section* directive is incorrect

bad #pragma space directive - syntax for the *#pragma space* directive is incorrect

bad #undef syntax - *#undef* is not followed by an identifier

bad _asm() argument type - the first argument passed to *_asm* is missing or is not a character string

bad alias expression - alias definition is not a valid expression

bad alias value - alias definition is not a constant expression

bad bit number - a bit number is not a constant between 0 and 7

bad character <character> - *<character>* is not part of a legal token

bad defined syntax - the *defined* operator must be followed by an identifier, or by an identifier enclosed in parenthesis

bad function declaration - function declaration has not been terminated by a right parenthesis

bad integer constant - an invalid integer constant has been specified

bad invocation of macro <name> - a *#define* macro defined without arguments has been invoked with arguments

bad macro argument - a parameter in a *#define* macro is not an identifier

bad macro argument syntax - parameters in a *#define* macro are not separated by commas

bad proto argument type - function prototype argument is declared without an explicit type

bad real constant - an invalid real constant has been specified

bad return type for inline function - inline function must be declared with *void* return type

bad space modifier - a modifier beginning with a *@* character is not followed by an identifier

bad structure for return - the structure for return is not compatible with that of the function

bad struct/union operand - a structure or an union has been used as operand for an arithmetic operator

bad symbol definition - the syntax of a symbol defined by the **-d** option on the command line is not valid

bad void argument - the type *void* has not been used alone in a prototyped function declaration

can't create <name> - file <name> cannot be created for writing

can't open <name> - file <name> cannot be opened for reading

can't redefine macro <name> - macro <name> has been already defined

can't undef macro <name> - a *#undef* has been attempted on a predefined macro

compare out of range - a comparison is detected as being always true or always false (**+strict**)

const assignment - a *const* object is specified as left operand of an assignment operator

constant assignement in a test - an assignment operator has been used in the test expression of an *if*, *while*, *do*, *for* statements or a conditional expression (+strict)

duplicate case - two *case* labels have been defined with the same value in the same *switch* statement

duplicate default - a *default* label has been specified more than once in a *switch* statement

embedded usage of tag name <name> - a structure/union definition contains a reference to itself.

enum size unknow - the range of an enumeration is not available to choose the smallest integer type

exponent overflow in real - the exponent specified in a real constant is too large for the target encoding

float value too large for integer cast - a float constant is too large to be casted in an integer

hexadecimal constant too large - an hexadecimal constant is too large to be represented on an integer

illegal storage class - storage class is not legal in this context

illegal type specification - type specification is not recognizable

illegal void operation - an object of type *void* is used as operand of an arithmetic operator

illegal void usage - an object of type *void* is used as operand of an assignment operator

implicit int type in argument declaration - an argument has been declared without any type (+strict)

implicit int type in global declaration - a global variable has been declared without any type (**+strict**)

implicit int type in local declaration - a local variable has been declared without any type (**+strict**)

implicit int type in struct/union declaration - a structure or union field has been declared without any type (**+strict**)

incompatible argument type - the actual argument type does not match the corresponding type in the prototype

incompatible compare type - operands of comparison operators must be of scalar type

incompatible operand types - the operands of an arithmetic operator are not compatible

incompatible pointer assignment - assigned pointers must have the same type, or one of them must be a pointer to *void*

incompatible pointer operand - a scalar type is expected when operators `+=` and `-=` are used on pointers

incompatible pointer operation - pointers are not allowed for that kind of operation

incompatible pointer types - the pointers of the assignment operator must be of equal or coercible type

incompatible return type - the return expression is not compatible with the declared function return type

incompatible struct/union operation - a structure or an union has been used as operand of an arithmetic operator

incompatible types in struct/union assignment - structures must be compatible for assignment

incomplete #elif expression - a *#elif* is followed by an incomplete expression

incomplete #if expression - a *#if* is followed by an incomplete expression

incomplete type - structure type is not followed by a tag or definition

incomplete type for debug information - a structure or union is not completely defined in a file compiled with the debug option set

integer constant too large - a decimal constant is too large to be represented on an integer

invalid case - a *case* label has been specified outside of a *switch* statement

invalid default - a *default* label has been specified outside of a *switch* statement

invalid ? test expression - the first expression of a ternary operator (*? :*) is not a testable expression

invalid address operand - the “address of” operator has been applied to a *register* variable or an rvalue expression

invalid address type - the “address of” operator has been applied to a bitfield

invalid alias - an alias has been applied to an *extern* object

invalid arithmetic operand - the operands of an arithmetic operator are not of the same or coercible types

invalid array dimension - an array has been declared with a dimension which is not a constant expression

invalid binary number - the syntax for a binary constant is not valid

invalid bit assignment - the expression assigned to a bit variable must be scalar

invalid bit initializer - the expression initializing a bit variable must be scalar

invalid bitfield size - a bitfield has been declared with a size larger than its type size

invalid bitfield type - a type other than *int*, *unsigned int*, *char*, *unsigned char* has been used in a bitfield.

invalid break - a break may be used only in *while*, *for*, *do*, or *switch* statements

invalid case operand - a case label has to be followed by a constant expression

invalid cast operand - the operand of a *cast* operator is not an expression

invalid cast type - a cast has been applied to an object that cannot be coerced to a specific type

invalid conditional operand - the operands of a conditional operator are not compatible

invalid constant expression - a constant expression is missing or is not reduced to a constant value

invalid continue - a continue statement may be used only in *while*, *for*, or *do* statements

invalid do test type - the expression of a *do ... while()* instruction is not a testable expression

invalid expression - an incomplete or ill-formed expression has been detected

invalid external initialization - an external object has been initialized

invalid floating point operation - an invalid operator has been applied to floating point operands

invalid for test type - the second expression of a *for(;;)* instruction is not a testable expression

invalid function member - a function has been declared within a structure or an union

invalid function type - the function call operator `()` has been applied to an object which is not a function or a pointer to a function

invalid if test type - the expression of an `if ()` instruction is not a testable expression

invalid indirection operand - the operand of unary `*` is not a pointer

invalid line number - the first parameter of a `#line` directive is not an integer

invalid local initialization - the initialization of a local object is incomplete or ill-formed

invalid lvalue - the left operand of an assignment operator is not a variable or a pointer reference

invalid narrow pointer cast - a cast operator is attempting to reduce the size of a pointer

invalid operand type - the operand of a unary operator has an incompatible type

invalid pointer cast operand - a cast to a function pointer has been applied to a pointer that is not a function pointer

invalid pointer initializer - initializer must be a pointer expression or the constant expression `0`

invalid pointer operand - an expression which is not of integer type has been added to a pointer

invalid pointer operation - an illegal operator has been applied to a pointer operand

invalid pointer types - two incompatible pointers have been subtracted

invalid shift count type - the right expression of a shift operator is not an integer

invalid sizeof operand type - the *sizeof* operator has been applied to a function

invalid storage class - storage class is not legal in this context

invalid struct/union operation - a structure or an union has been used as operand of an arithmetic operator

invalid switch test type - the expression of a *switch ()* instruction must be of integer type

invalid typedef usage - a typedef identifier is used in an expression

invalid void pointer - a *void* pointer has been used as operand of an addition or a subtraction

invalid while test type - the expression of a *while ()* instruction is not a testable expression

missing ## argument in macro <name> - an argument of a *##* operator in a *#define* macro is missing

missing '>' in #include - a file name of a *#include* directive begins with '<' and does not end with '>'

missing) in defined expansion - a '(' does not have a balancing ')' in a *defined* operator

missing ; in argument declaration - the declaration of a function argument does not end with ';'

missing ; in local declaration - the declaration of a local variable does not end with ';'

missing ; in member declaration - the declaration of a structure or union member does not end with ';'.

missing ? test expression - the test expression is missing in a ternary operator (*? :*)

missing _asm() argument - the *_asm* function needs at least one argument

missing argument - the number of arguments in the actual function call is less than that of its prototype declaration

missing argument for macro <name> - a macro invocation has fewer arguments than its corresponding declaration

missing argument name - the name of an argument is missing in a prototyped function declaration

missing array subscript - an array element has been referenced with an empty subscript

missing do test expression - a *do ... while ()* instruction has been specified with an empty *while* expression

missing enumeration member - a member of an enumeration is not an identifier

missing explicit return - a return statement is not ending a non-void function (**+strict**)

missing exponent in real - a floating point constant has an empty exponent after the 'e' or 'E' character

missing expression - an expression is needed, but none is present

missing file name in #include - a *#include* directive is used, but no file name is present

missing goto label - an identifier is needed after a *goto* instruction

missing if test expression - an *if ()* instruction has been used with an empty test expression

missing initialization expression - a local variable has been declared with an ending '=' character not followed by an expression

missing initializer - a simple object has been declared with an ending '=' character not followed by an expression

missing local name - a local variable has been declared without a name

missing member declaration - a structure or union has been declared without any member

missing member name - a structure or union member has been declared without a name

missing name in declaration - a variable has been declared without a name

missing prototype - a function has been used without a fully prototyped declaration (**+strict**)

missing prototype for inline function - an inline function has been declared without a fully prototyped syntax

missing return expression - a simple return statement is used in a non-void function (**+strict**)

missing switch test expression - an expression in a *switch* instruction is needed, but is not present

missing while - a *'while'* is expected and not found

missing while test expression - an expression in a *while* instruction is needed, but none is present

missing : - a *':'* is expected and not found

missing ; - a *','* is expected and not found

missing (- a *'('* is expected and not found

missing) - a *'),'* is expected and not found

missing] - a *']'* is expected and not found

missing { - a *'{'* is expected and not found

missing } - a *'}'* is expected and not found

missing } in enum definition - an enumeration list does not end with a `}` character

missing } in struct/union definition - a structure or union member list does not end with a `}` character

redeclared argument <name> - a function argument has conflicting declarations

redeclared enum member <name> - an *enum* element is already declared in the same scope

redeclared external <name> - an *external* object or function has conflicting declarations

redeclared local <name> - a *local* is already declared in the same scope

redeclared proto argument <name> - an identifier is used more than once in a prototype function declaration

redeclared typedef <name> - a *typedef* is already declared in the same scope

redefined alias <name> - an *alias* has been applied to an already declared object

redefined label <name> - a *label* is defined more than once in a function

redefined member <name> - an identifier is used more than once in structure member declaration

redefined tag <name> - a *tag* is specified more than once in a given scope

repeated type specification - the same type modifier occurs more than once in a type specification

scalar type required - type must be integer, floating, or pointer

size unknown - an attempt to compute the size of an unknown object has occurred

space attribute conflict - a space modifier attempts to redefine an already specified modifier

string too long - a string is used to initialize an array of characters shorter than the string length

struct/union size unknown - an attempt to compute a structure or union size has occurred on an undefined structure or union

syntax error - an unexpected identifier has been read

token overflow - an expression is too complex to be parsed

too many argument - the number of actual arguments in a function declaration does not match that of the previous prototype declaration

too many arguments for macro <name> - a macro invocation has more arguments than its corresponding macro declaration

too many initializers - initialization is completed for a given object before initializer list is exhausted

too many spaces modifiers - too many different names for '@' modifiers are used

truncating assignment - the right operand of an assignment is larger than the left operand (**+strict**)

unbalanced ' - a character constant does not end with a single quote

unbalanced " - a string constant does not end with a double quote

<name> undefined - an undeclared identifier appears in an expression

undefined label <name> - a label is never defined

undefined struct/union - a structure or union is used and is never defined

unexpected end of file - last declaration is incomplete

unexpected return expression - a return with an expression has been used within a *void* function

unknown enum definition - an enumeration has been declared with no member

unknown structure - an attempt to initialize an undefined structure has been done

unknown union - an attempt to initialize an undefined union has been done

value out of range - a constant is assigned to a variable too small to represent its value (**+strict**)

zero divide - a divide by zero was detected

zero modulus - a modulus by zero was detected

Code Generator (cgxgate) Error Messages

bad builtin - the *@builtin* type modifier can be used only on functions

bad @interrupt usage - the *@interrupt* type modifier can only be used on functions.

invalid indirect call - a function has been called through a pointer with more than one *char* or *int* argument, or is returning a structure.

redefined space - the version of *cp* you used to compile your program is incompatible with *cg*.

unknown space - you have specified an invalid space modifier *@xxx*

unknown space modifier - you have specified an invalid space modifier *@xxx*

PANIC ! bad input file - cannot read input file

PANIC ! bad output file - cannot create output file

PANIC ! can't write - cannot write output file

All other **PANIC !** messages should never happen. If you get such a message, please report it with the corresponding source program to COSMIC.

Assembler (caxgate) Error Messages

The following error messages may be generated by the assembler. Note that the assembler's input is machine-generated code from the compiler. Hence, it is usually impossible to fix things 'on the fly'. The problem must be corrected in the source, and the offending program(s) recompiled.

bad .source directive - a *.source* directive is not followed by a string giving a file name and line numbers

bad addressing mode - an invalid addressing mode have been constructed

bad argument number- a parameter sequence *\n* uses a value negative or greater than 9

bad character constant - a character constant is too long for an expression

bad comment delimiter- an unexpected field is not a comment

bad constant - a constant uses illegal characters

bad else - an *else* directive has been found without a previous *if* directive

bad endif - an *endif* directive has been found without a previous *if* or *else* directive

bad file name - the *include* directive operand is not a character string

bad index register - an invalid register has been used in an indexed addressing mode

bad register - an invalid register has been specified as operand of an instruction

bad relocatable expression - an external label has been used in either a constant expression, or with illegal operators

bad string constant - a character constant does not end with a single or double quote

bad symbol name: <name> - an expected symbol is not an identifier

can't create <name> - the file <name> cannot be opened for writing

can't open <name> - the file <name> cannot be opened for reading

can't open source <name> - the file <name> cannot be included

cannot include from a macro - the directive *include* cannot be specified within a macro definition

cannot move back current pc - an *org* directive has a negative offset

illegal size - the size of a *ds* directive is negative or zero

missing label - a label must be specified for this directive

missing operand - operand is expected for this instruction

missing register - a register is expected for this instruction

missing string - a character string is expected for this directive

relocatable expression not allowed - a constant is needed

section name <name> too long - a section name has more than 15 characters

string constant too long - a string constant is longer than 255 characters

symbol <name> already defined - attempt to redefine an existing symbol

symbol <name> not defined - a symbol has been used but not declared

syntax error - an unexpected identifier or operator has been found

too many arguments - a macro has been invoked with more than 9 arguments

too many back tokens - an expression is too complex to be evaluated

unclosed if - an *if* directive is not ended by an *else* or *endif* directive

unknown instruction <name> - an instruction not recognized by the processor has been specified

value too large - an operand is too large for the instruction type

zero divide - a divide by zero has been detected

Linker (clnk) Error Messages

-a not allowed with -b or -o - the *after* option cannot be specified if any start address is specified.

+def symbol <symbol> multiply defined - the symbol defined by a *+def* directive is already defined.

bad file format - an input file has not an object file format.

bad number in +def - the number provided in a *+def* directive does not follow the standard C syntax.

bad number in +spc <segment> - the number provided in a *+spc* directive does not follow the standard C syntax.

bad processor type - an object file has not the same configuration information than the others.

bad reloc code - an object file contains unexpected relocation information.

bad section name in +def - the name specified after the '@' in a *+def* directive is not the name of a segment.

bank crossing call - a *jsr* instruction has been used to enter a banked function, either from a different bank or from a common area.

can't create map file <file> - map file cannot be created.

can't create <file> - output file cannot be created.

can't locate .text segment for initialization - initialized data segments have been found but no host segment has been specified.

can't locate shared segment - shared datas have been found but no host segment has been specified.

can't open file <file> - input file cannot be found.

file already linked - an input file has already been processed by the linker.

function <function> is recursive - a *nostack* function has been detected as recursive and cannot be allocated.

function <function> is reentrant - a function has been detected as reentrant. The function is both called in an interrupt function and in the main code.

incomplete +def directive - the **+def** directive syntax is not correct.

incomplete +seg directive - the **+seg** directive syntax is not correct.

incomplete +spc directive - the **+spc** directive syntax is not correct.

init segment cannot be initialized - the host segment for initialization cannot be itself initialized.

invalid @ argument - the syntax of an optional input file is not correct.

invalid -i option - the **-i** directive is followed by an unexpected character.

missing command file - a link command file must be specified on the command line.

missing output file - the **-o** option must be specified.

missing '=' in +def - the **+def** directive syntax is not correct.

missing '=' in +spc <segment> - the **+spc** directive syntax is not correct.

named segment <segment> not defined - a segment name does not match already existing segments.

no default placement for segment <segment> - a segment is missing **-a** or **-b** option.

prefixed symbol <name> in conflict - a symbol beginning by 'f_' (for a banked function) also exists without the 'f' prefix.

read error - an input object file is corrupted

segment <segment> and <segment> overlap - a segment is overlapping an other segment.

segment <segment> size overflow - the size of a segment is larger than the maximum value allowed by the **-m** option.

shared segment not empty - the host segment for shared data is not empty and cannot be used for allocation.

symbol <symbol> multiply defined - an object file attempts to redefine a symbol.

symbol <symbol> not defined - a symbol has been referenced but never defined.

unexpected bank location - an interrupt function or a function accessing the PPAGE register is located in a bank.

unknown directive - a directive name has not been recognized as a linker directive.

Modifying Compiler Operation

This chapter tells you how to modify compiler operation by making changes to the standard configuration file. It also explains how to create your own “programmable options” which you can use to modify compiler operation from the [cxxgate.cxf](#).

The Configuration File

The configuration file is designed to define the default options and behaviour of the compiler passes. It will also allow the definition of programmable options thus simplifying the compiler configuration. A configuration file contains a list of options similar to the ones accepted for the compiler driver utility **cxgate**.

These options are described in **Chapter 4**, “[Using The Compiler](#)”. There are two differences: the option **-f** cannot be specified in a configuration file, and the extra **-m** option has been added to allow the definition of a programmable compiler option, as described in the next paragraph.

The contents of the configuration file **cxgate.cxf** as provided by the default installation appears below:

```
# CONFIGURATION FILE FOR XGATE COMPILER
# Copyright (c) 2004 by COSMIC Software
#
-pu                # unsigned char
-i c:\cx32\hxgate  # include path
#
-m debug:x         # debug: produce debug info
-m nobss:,bss      # nobss: do not use bss
-m nocst:,ct       # nocst: constant in text section
-m nofr:,nf        # nofr: no interrupt argument frame
-m proto:p         # proto: enable prototype checking
-m rev:rb          # rev: reverse bit field order
-m v1:,sv1         # v1: first version of silicon
-m warn:w1         # warn: enable warnings
```

The following command line:

```
cxxgate hello.c
```

in combination with the above configuration file directs the **cxxgate** compiler to execute the following commands:

```
cpxgate -o \2.cx1 -u -i\cx32\hxgate hello.c
cgxgate -o \2.cx2 \2.cx1
coxgate -o \2.cx1 \2.cx2
caxgate -o hello.o -i\cx32\hxgate \2.cx1
```

Changing the Default Options

To change the combination of options that the compiler will use, edit the configuration file and add your specific options using the **-p** (for the **p**arser), **-g** (for the code **g**enerator), **-o** (for the **o**ptimizer) and **-a** (for the **a**sssembler) options. If you specify an invalid option or combination of options, compilation will not proceed beyond the step where the error occurred. You may define up to 60 such options.

Creating Your Own Options

To create a programmable option, edit the configuration file and define the parametrable option with the **-m*** option. The string ***** has the following format:

```
name:popt,gopt,oopt,aopt,exclude...
```

The first field defines the option *name* and must be ended by a colon character **:**. The four next fields describe the effect of this option on the four passes of the compiler, respectively the *parser*, the *generator*, the *optimizer* and the *assembler*. These fields are separated by a comma character **,**. If no specific option is needed on a pass, the field has to be specified empty. The remaining fields, if specified, describe a exclusive relationship with other defined options. If two *exclusive* options are specified on the command line, the compiler will stop with an error message. You may define up to 20 programmable options. At least one field has to be specified. Empty fields need to be specified only if a useful field has to be entered after.

In the following example:

```
-m dl1:1,dl1,,,dl2# dl1: line option 1
-m dl2:1,dl2,,,dl1# dl1: line option 2
```

the two options *dl1* and *dl2* are defined. If the option **+dl1** is specified on the compiler command line, the specific option **-l** will be used for the *parser* and the specific option **-dl1** will be used for the code *generator*. No specific option will be used for the *optimizer* and for the *assembler*. The option *dl1* is also declared to be exclusive with the option *dl2*, meaning that *dl1* and *dl2* will not be allowed together on the compiler command line. The option *dl2* is defined in the same way.

Example

The following command line

```
cxxgate +nobss hello.c
```

in combination with the previous configuration file directs the **cxxgate** compiler to execute the following commands:

```
cpxgate -o \2.cx1 -u -i\cx32\hxgate hello.c
cgxgate -o \2.cx2 -bss \2.cx1
coxgate -o \2.cx1 \2.cx2
caxgate -o hello.o -i\cx32\hxgate \2.cx1
```

XGATE Machine Library

This appendix describes each of the functions in the Machine Library (**libm**). These functions provide the interface between the XGATE microcontroller hardware and the functions required by the code generator. They are described in reference form, and listed alphabetically.

Function Listing

x_fadd:	float addition
x_fcmp:	float comparison
x_fdiv:	float division
x_fmul:	float multiplication
x_fsub:	float subtraction
x_ftol2:	float to long conversion in R2/R3 registers
x_ftol4:	float to long conversion in R4/R5 registers
x_idiv:	integer division
x_imul:	integer multiplication
x_jitab:	int direct switch
x_jltab:	long switch
x_jtab:	integer switch
x_ldiv:	long divide
x_lmul:	long multiplication

x_ltof2: long to float conversion in R2/R3 registers
x_ltof4: long to float conversion in R4/R5 registers
x_ludv: long unsigned divide
x_udiv: integer unsigned division
x_ultof2: unsigned long to float conversion in R2/R3 registers
x_ultof4: unsigned long to float conversion in R4/R5 registers

APPENDIX D

Compiler Passes

The information contained in this appendix is of interest to those users who want to modify the default operation of the cross compiler by changing the configuration file that the **cxsgate** compiler uses to control the compilation process.

This appendix describes each of the passes of the compiler:

cpxgate	the parser
cgxgate	the code generator
coxgate	the assembly language optimizer

The cpxgate Parser

cpxgate is the parser used by the C compiler to expand *#defines*, *#includes*, and other directives signalled by a *#*, parse the resulting text, and outputs a sequential file of flow graphs and parse trees suitable for input to the code generator **cgxgate**.

Command Line Options

cpxgate accepts the following options, each of which is described in detail below:

```
cpxgate [options] file
  -ad      expand defines in assembly
  -c99     c99 type behaviour
  -ck      extra type checkings
  -cp      no constant propagation
  -d*>     define symbol=value
  -e       run preprocessor only
  +e*      error file name
  -h*>     include header
  -i*>     include path
  -l       output line information
  -m#      model configuration
  -nc      no const replacement
  -ne      no enum optimization
  -np      allow pointer narrowing
  -o*      output file name
  -p       need prototypes
  -rb      reverse bitfield order
  -s       do not reorder locals
  -sa      strict ANSI conformance
  -u       plain char is unsigned
  -w#      enable warnings
  -xd      debug info for data
  -xp      no path in debug info
  -xx      extended debug info
  -x       output debug info
```

-ad enable #define expansion inside inline assembly code between *#asm* and *#endasm* directives. By default, #define symbols are expanded only in the C code.

- c99** authorize the repetition of the `const` and `volatile` modifiers in the declaration either directly or indirectly in the `type-def`.
- ck** direct the compiler to enforce stronger type checking.
- cp** disable the constant propagation optimization. By default, when a variable is assigned with a constant, any subsequent access to that variable is replaced by the constant itself until the variable is modified or a flow break is encountered (function call, loop, label ...).
- d*^** specify `*` as the name of a user-defined preprocessor symbol (`#define`). The form of the definition is **-dsymbol[=value]**; the symbol is set to `1` if value is omitted. You can specify up to 60 such definitions.
- e** run preprocessor only. *cpxgate* only outputs lines of text.
- +e*** log errors in the text file `*` instead of displaying the messages on the terminal screen.
- h*>** include files before to start the compiler process. You can specify up to 60 files.
- i*>** specify include path. You can specify up to 60 different paths. Each path is a directory name, **not** terminated by any directory separator character.
- l** output line number information for listing or debug.
- m#** the value `#` is used to configure the parser behaviour. It is a two bytes value, the upper byte specifies the default space for variables, and the lower byte specifies the default space for functions. A space byte is the or'ed value between a size specifier and several optional other specifiers. The allowed size specifiers are:

0x10	@tiny
-------------	-------

0x20	@near
0x30	@far

Allowed optionals specifiers are:

0x02	@pack
0x04	@nostack

Note that all the combinations are not significant for all the target processors.

- nc** do not replace an access to an initialized const object by its value. By default, the usage of a const object whose value is known is replaced by its constant value.
- ne** do not optimize size of *enum* variables. By default, the compiler selects the smallest integer type by checking the range of the declared *enum* members. This mechanism does not allow uncomplete *enum* declaration. When the **-ne** option is selected, all *enum* variables are allocated as *int* variables, thus allowing uncomplete declarations, as the knowledge of all the members is no more necessary to choose the proper integer type.
- np** allow pointer narrowing. By default, the compiler refuses to cast the pointer into any smaller object. This option should be used carefully as such conversions are truncating addresses.
- o*** write the output to the file *. Default is STDOUT for output if **-e** is specified. Otherwise, an output file name is required.
- p** enforce prototype declaration for functions. An error message is issued if a function is used and no prototype declaration is found for it. By default, the compiler accepts both syntaxes without any error.

- rb** reverse the bitfield fill order. By default, bitfields are filled from **less** significant bit (**LSB**) to **most** significant bit (**MSB**). If this option is specified, filling works from most significant bit to less significant bit.
- s** do not reorder local variables. By default, the compiler sorts the local variables of a function in order to allocate the most used variables as close as possible to the frame pointer. This allows to use the shortest addressing modes for the most used variables.
- sa** enforce a strict ANSI checking by rejecting any syntax or semantic extension. This option also disables the enum size optimization (**-ne**).
- u** take a plain char to be of type **unsigned char**, not signed char. This also affects in the same way strings constants.
- w#** enable warnings if # is greater or equal to 0. By default, warnings are disabled.
- x** generate debugging information for use by the cross debugger or some other debugger or in-circuit emulator. The default is to generate no debugging information.
- xd** add debug information in the object file only for data objects, hiding any function.
- xp** do not prefix filenames in the debug information with any absolute path name. Debuggers will have to be informed about the actual files location.
- xx** add debug information in the object file for any label defining code or data.

Return Status

cpxgate returns success if it produces no error diagnostics.

Example

cpxgate is usually invoked before *cgxgate* the code generator, as in:

```
cpxgate -o \2.cx1 -u -i \cosmic\hxgate file.c  
cgxgate -o \2.cx2 \2.cx1
```

The *cgxgate* Code Generator

cgxgate is the code generating pass of the C compiler. It accepts a sequential file of flow graphs and parse trees from *cpngxgate* and outputs a sequential file of assembly language statements.

As much as possible, the compiler generates freestanding code, but, for those operations which cannot be done compactly, it generates inline calls to a set of machine-dependent runtime library routines.

Command Line Options

cgxgate accepts the following options, each of which is described in detail below:

```
cgxgate [options] file
-a      optimize _asm code
-bss    do not use bss
-ct     constants in code
-dl#    output line information
+e*     error file name
-f      full source display
-l      output listing
-na     do not xdef alias name
-nf     no frame argument
-no     do not use optimizer
-o*     output file name
-st*    user stack name
-sv#    silicon version
-v      verbose
```

- a** optimize *_asm* code. By default, the assembly code inserted by a *_asm* call is left unchanged by the optimizer.
- bss** inhibit generating code into the *bss* section.
- ct** output constant in the *.text* section. By default, the compiler outputs literals and constants in the *.const* section.
- dl#** produce line number information. # must be either '1' or '2'. Line number information can be produced in two ways: 1) function name and line number is obtained by

specifying **-dl1**; 2) file name and line number is obtained by specifying **-dl2**. All information is coded in symbols that are in the debug symbol table.

- +e*** log errors in the text file *** instead of displaying the messages on the terminal screen.
- f** merge all C source lines of functions producing code into the C and Assembly listing. By default, only C lines actually producing assembly code are shown in the listing.
- l** merge C source listing with assembly language code; listing output defaults to *<file>.ls*.
- nf** uses **R1** as the only argument of interrupt functions. By default, **R1** is handled as a frame pointer and allows several arguments to be passed to interrupt functions.
- no** do not produce special directives for the post-optimizer.
- o*** write the output to the file *** and write error messages to STDOUT. The default is STDOUT for output and STDERR for error messages.
- st*** specify the symbol name of the stack. This symbol is usually defined in the linker command file. The default stack pointer symbol name is **__xstack**.
- sv#** specify the silicon version of the target processor. Default value is 2.
- v** When this option is set, each function name is send to STDERR when *cgxgate* starts processing it.

Return Status

cgxgate returns success if it produces no diagnostics.

Example

cgxgate usually follows *cpkgate* as follows:

```
cpxgate -o \2.cx1 -u -i\cosmic\hxgate file.c  
cgxgate -o \2.cx2 \2.cx1
```

The coxgate Assembly Language Optimizer

coxgate is the code optimizing pass of the C compiler. It reads source files of **XGATE** assembly language source code, as generated by the **cxgate** code generator, and writes assembly language statements. **coxgate** is a peephole optimizer; it works by checking lines function by function for specific patterns. If the patterns are present, **coxgate** replaces the lines where the patterns occur with an optimized line or set of lines. It repeatedly checks replaced patterns for further optimizations until no more are possible. It deals with redundant load/store operations, constants, stack handling, and other operations.

Command Line Options

coxgate accepts the following options, each of which is described in detail below:

```
coxgate [options] <file>
  -c      keep original lines as comments
  -d*     disable specific optimizations
  -o*     output file name
  -v      print efficiency statistics
```

- c** leave removed instructions as comments in the output file.
- d*** specify a list of codes allowing specific optimizations functions to be selectively disabled.
- o*** write the output to the file * and write error messages to STDOUT. The default is STDOUT for output and STDERR for error messages.
- v** write a log of modifications to STDERR. This displays the number of removed instructions followed by the number of modified instructions.

If *<file>* is present, it is used as the input file instead of the default STDIN.

Disabling Optimization

When using the optimizer with the **-c** option, lines which are changed or removed are kept in the assembly source as comment, followed by a code composed with a letter and a digit, identifying the internal function which performs the optimization. If an optimization appears to do something wrong, it is possible to disable selectively that function by specifying its code with the **-d** option. Several functions can be disabled by specifying a list of codes without any whitespaces. The code letter can be enter both lower or uppercase.

Return Status

coxgate returns success if it produces no diagnostics.

Example

coxgate is usually invoked after *cgxgate* as follows:

```
cpxgate -o \2.cx1 -u -i\cosmic\hxgate file.c
cgxgate -o \2.cx2 \2.cx1
coxgate -o file.s \2.cx2
```


Index

Symbols

- #asm
 - directive 278
- #asm directive 30
- #define
 - replacement in assembly 31
- #endasm
 - directive 278
- #endasm directive 30
- #pragma asm directive 30
- #pragma endasm directive 30
- .const
 - output section 283
- .xbss
 - section 27
- .xconst
 - section 38
- .xdata
 - section 27
- .xtext
 - section 38
- @interrupt
 - qualifier 34
- __xstack
 - symbol 19, 34, 243, 284
 - symbol value 34
- _asm 51
 - argument string size 31
 - assembly sequence 32
 - code optimization 283
 - lowercase mnemonics 32
 - return type 33

- uppercase mnemonics 32

- _asm() function 55
- _bfft function 65
- _carry function 67
- _csem function 71
- _par function 110
- _sif function 128
- _ssem function 135

Numerics

- 8-bit precision, operation 10

A

- abort function 56
- abs function 57
- absolute
 - map section 166
 - path name 281
 - section, org 223
- absolute addresses 28
- absolute section 232
- absolute value, find 57
- acos function 58
- address
 - extension, page 175
- align directive 182
- Arcosine 58
- Arcsine 59
- Arctangent 60
- Arctangent of y/x 61
- argument
 - output format 112

- widening 156
- asin function 59
- assembler
 - branch shortening 180
 - C style directives 181
 - code inline 31
 - conditional branch range 180
 - conditional directive 178
 - create listing file 167
 - cross-reference 169
 - environment symbol 179
 - error for undefined symbol 168
 - expression 173
 - generate
 - listing file 168
 - object file 168
 - label 171
 - macro
 - directive 175
 - endm directive 175
 - instruction 175
 - operator set 174
 - sections 179
 - switch directive 179
- assembly language
 - code optimizer 286
- atan function 60
- atan2 function 61
- atof function 62
- atoi function 63
- atol function 64

B

- bank
 - page operator 175
- base directive 183
- bitfield
 - compiler reverse option 48
 - filling 281
 - filling order 48
 - reverse order 281
- bsct directive 184

- buffer to double 62
- buffer to integer 63
- buffer to long 64

C

- C interface
 - to assembly language 38
- C library
 - floating point functions 52
 - integer functions 51
 - package 51
- C source
 - lines merging 284
- calling environment 99
- calloc function 66
- ceil function 68
- char
 - signed 281
 - unsigned 281
- characters, replace 145
- clist directive 185, 200, 202, 203, 204, 205, 206, 207, 208, 209, 210
- code generator
 - compiler pass 283
 - error log file 284
- code optimizer
 - compiler pass 286
- compiler
 - ANSI checking 281
 - assembler 9
 - assembler option specification 45
 - C preprocessor and language parser 8
 - code generation option specification 46
 - code generator 8
 - code optimization 10
 - code optimizer 8
 - combination of options 273
 - command line option 44
 - configuration file 272
 - configuration file specification 46
 - constant in code section 48

-
- create assembler file only 47
 - default behavior 44
 - default configuration file 46
 - default file names 49
 - default operations 277
 - default options 44, 272
 - driver 4
 - error file path specification 46
 - error log file 46
 - error message 44
 - exclusive options 273
 - flags 6
 - generate error 249
 - generate error file 50
 - generate listing 50
 - include path definition 47
 - invoke 44
 - listing file 47
 - listing file path specification 46
 - name 44
 - object file path specification 46
 - optimizer option specification 47
 - options 44
 - options request 44
 - parser option specification 47
 - predefined option selection 47
 - preprocessed file only 47
 - produce debug information 48
 - programmable option 272, 273
 - specific options 4
 - specify options 45
 - temporary files path 47
 - type checking 279
 - user-defined preprocessor symbol 46
 - verbose 47
 - verbose mode 16
 - widen to int 156
 - configuration file
 - predefined options 47
 - const
 - qualifier 25
 - constant
 - in .text section 283
 - numeric 172
 - prefix character 172
 - string 172
 - string character 173
 - suffix character 173
 - control character 84
 - convert 62, 63, 64
 - copy from one buffer to another 105, 106
 - cos function 69
 - cosh function 70
 - cosine 69
 - cross-reference
 - information 166
 - output 17
- ## D
- data
 - const type 26
 - initialized 27
 - volatile type 25
 - data representation
 - @tiny pointers 42
 - float and double 42
 - long int, 32 bit pointer 42
 - short int, 16 bit pointer 42
 - dc directive 186
 - dcb directive 187
 - debug information
 - add line 168
 - adding 281
 - label 168
 - no prefix 168
 - debug symbol
 - in object file 168
 - debugging information
 - generate 281
 - default
 - bitfield order 281
 - branch optimization 166
 - output file 167
 - div function 72

dlist directive 188

ds directive 189

E

else directive 190, 191, 194, 200, 202, 208

end directive 192

end5 directive 196

endc directive 202, 208

endif directive 190, 193, 194, 200

endm directive 195, 215, 218, 230

endr 226, 227

enum

size optimization 280

equ directive 197, 234

error

assembler log file 167

file name 50

message 10

message list 249

multiply defined symbol 171

even directive 198

exit 73

exp function 74

expression

evaluation 175

high 175

low 175

page 175

F

fabs function 75

fail directive 199

fill

byte 167, 223

filling byte 189, 198

find 61

floating point library 51

Floating Point Library Functions 52

floor function 76

fmod function 77

format

specifiers 112

formatted arguments, output to stdout 112

formatted string, conversion specifications 112

fraction and integer from double, extract 109

free function 78

frexp function 79

function

@inline modifier 36

enforce prototype declaration 48, 280

prototype declaration 48, 280

returning int 54

G

generate

.xbss section 38

.xconst section 38

.xdata section 38

.xtext section 38

listing file 17

output files 15

getchar function 80

gets function 81

H

header files 53

heap

allocate space 101

free space 78

heap space 66

-help option 6

I

IEEE Floating Point Standard 42

if

directive 178

if directive 190, 194, 200

if directive 193

ifc directive 201

ifdef directive 202

- ifeq directive 203
- ifge directive 204
- ifgt directive 205
- ifle directive 206
- iflt directive 207
- ifnc directive 210
- ifndef directive 208
- ifne directive 209
- include
 - assembler directive 179
 - assembly file 167
 - define path 167
 - file before 279
 - module 243
 - path specification 279
 - specify path 279
- include directive 211
- inline
 - #pragma directive 30
 - assembly code 31
 - assembly instruction 30
 - block inside a function 30
 - block outside a function 30
 - function 36
 - header function 53
 - with _asm function 31, 32
 - with pragma sequences 30
- inline function
 - _bfft 37
 - _carry 37
 - _csem 37
 - _par 36
 - _sif 36
 - _ssem 37
- input
 - read 122
 - read from string 134
- input/output 26, 29
- input/output registers 29
- integer
 - library 243
- interrupt

- handler 34
 - handler address 35
 - return sequence 34
 - vector 35
- isalnum function 82
- isalpha function 83
- isctrl function 84
- isdigit function 85
- isgraph function 86
- islower function 87
- isprint function 88
- ispunct function 89
- isqrt function 90
- isspace function 91
- isupper function 92
- isxdigit function 93

L

- label
 - temporary, local directive 172
- labs function 94
- ldexp function 95
- ldiv function 96
- library
 - file 243
 - floating point 51
 - integer 51, 243
 - machine 51
 - single precision 243
 - Standard ANSI 243
 - version 243
- line number
 - information 283
- link
 - relocatable file 17
- link command file 18
- linker
 - clnk 9
- list directive 212
- listing
 - absolute information 169
 - file location 21

- interspersed C and assembly file 16
- stream 169
- lit directive 213
- local
 - assembler directive 214
 - labels 33
- local variable
 - reorder 281
- log function 97
- log10 function 98
- logarithm 97
- longjmp function 99

M

- macro
 - argument 176
 - directive 215
 - expansion 31
 - internal labels 171
 - named syntax 177
 - named syntax, example 216
 - numbered syntax 176
 - numbered syntax, example 216
 - parameter 176
 - special parameter \# 176
 - special parameter * 177
 - special parameter \0 177, 216
- malloc function 101
- max function 102
- memchr function 103
- memcmp function 104
- memcpy function 105
- memmove function 106
- memory
 - allocate 121
- memory location 28
- memory mapped I/O 28
- memory mapped I/O port 29
- memset function 107
- messg directive 217
- mexit directive 216, 218
- min function 108

- mlist directive 219
- modf function 109
- Motorola
 - assembler syntax 170
 - old syntax option 167

N

- natural 97
- nolist directive 220
- nopage directive 221

O

- object
 - file location 21
 - relocatable file output 168
- object code output 168
- offset
 - assembler directive 222
 - start absolute section 222
- optimization
 - disable selectively 287
 - keep line 287
 - specific code 286
- optimizer
 - disable 47
- org directive 223
- output
 - formatted argument to buffer 162, 163
 - listing line number 279
 - to buffer, formatted argument 131

P

- page
 - value 175
- page directive 224
- parser
 - behaviour 279
 - compiler pass 278
 - error log file 279
- phase angle of a vector 61
- Plain pointers 42

- plen directive 225
- pointer
 - narrow 280
- positive integer 68
- pow function 111
- prefix
 - _ character 38
 - filename 281
- preprocessor
 - #define 278
 - #include 278
 - run only 279
- printf function 112
- pseudo-random number, generate 119
- pseudo-random number, seed 133
- putchar function 117
- puts function 118

R

- rand function 119
- realloc function 120
- relocation
 - physical 242
- repeat directive 226
- repeatl directive 227
- restore 99
- restore directive 229
- rexit directive 227, 230
- ROM 28
- round to 68

S

- save directive 231
- sbreak function 121
- scanf function 122
- section
 - .xbss 18, 27
 - .xconst 17
 - .xdata 18, 27
 - .xtext 17, 27
 - curly braces, initialized data 27
 - name 27, 179

- parenthesis, code 27
- pragma definition 27
- pragma directive 28
- square brackets, uninitialized data 27
- user defined 27
- xconst 27
- section directive 232
- sections
 - default 27
 - predefined 27
- set directive 234
- setjmp function 126
- silicon
 - first 48
 - second 48
- sin function 129
- sinh function 130
- space
 - for function 279
 - for variable 279
- space allocate 66
- space, reallocate on the heap 120
- spc directive 235
- sprintf function 131
- sqrt function 132
- square root
 - unsigned int compute 90
 - unsigned long int compute 100
- srand function 133
- sscanf function 134
- stack
 - free space 78
 - symbol name 284
- standard ANSI libraries 243
- strcat function 136
- strchr function 137
- strcmp function 138
- strcpy function 139
- strncpy function 140
- strings, copy n length 144
- strlen function 141
- strncat function 142

- strncmp function 143
- strncpy function 144
- strpbrk function 145
- strchr function 146
- strspn function 147
- strstr function 148
- strtod function 149
- strtol function 150
- strtoul function 151
- suffix
 - assembly file 44
 - C file 44
- switch directive 236
- symbol
 - user-defined 279

T

- tabs directive 237
- tan function 152
- tangent, compute 152
- tanh function 153
- test for 84
- title directive 238
- tolower function 154
- toupper function 155

U

- uninitialized variables 48
- unreachable code
 - eliminate 10

V

- va_arg macro 156
- va_end macro 158
- va_start macro 160
- variable
 - reorder local 281
- variable length argument list 158, 160
- volatile
 - data 25
 - memory mapped control registers 25
 - qualifier 25

- using keyword 25
- vprintf function 162
- vsprintf function 163

W

- warnings 48, 281

X

- x to the y power, compute 111
- xdef directive 239, 240
- XGATE
 - addressing mode 170
 - mnemonics 170
- xref directive 239, 240

Z

- zero page section 184