COSMIC Software

Version 4.1

Evaluation Version

C Cross Compiler User’s Guide for Motorola MC68HC05

Table of Contents

Chapter 5
Using The Assembler

Chapter D
Compiler Passes

Preface

The *Cross Compiler User's Guide for MC68HC05* is a reference guide for programmers writing C programs for MC68HC05 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 MC68HC05 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 MC68HC05 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.

Chapter 7, "*Debugging Support*", describes the support available for COSMIC's C source level cross debugger and for other debuggers or in-circuit emulators.

Chapter 8, "*Programming Support*", describes the programming support utilities. Examples of how to use these utilities are also included.

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, "*MC68HC05 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

The C cross compiler targeting the MC68HC05 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 **cx6805** 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 appears as:

```
cx6805 +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:

```
cx6805 +debug file.c
```

In this example, `cx6805 +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|h05]
```

means that one file with the extension `.o` or `.h05` 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:

-a	which accepts a long integer value,
-b	which accepts a long integer value,
-e	which accepts a long integer value,
-f	which accepts a single character,
-h	which is simply a flag indicator,

+h	which accepts a character string,
-m	which accepts a short integer value,
-n	which accepts a character string and may be repeated
-o	which accepts a character string.
-p	which is simply a flag indicator,
-pn	which is simply a flag indicator,
-pp	which is simply a flag indicator,
-s	which is simply a flag indicator and
-x	which accepts a character string

Compiler Architecture

The C compiler consists of several programs that work together to translate your C source files to executable files and listings. **cx6805** controls the operation of these programs automatically, using the options you specify, and runs the programs described below in the order listed:

cp6805 - the C preprocessor and language parser. *cp6805* expands directives in your C source and parses the resulting text.

cg6805 - the code generator. *cg6805* accepts the output of *cp6805* and generates assembly language statements.

co6805 - the assembly language optimizer. *co6805* optimizes the assembly language code that *cg6805* generates.

ca6805 - the assembler. *ca6805* converts the assembly language output of *co6805* 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 **cx6805**, 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 listed below to inspect the executable.

chex - absolute hex file generator. *chex* translates executable images produced by the linker into hexadecimal interchange formats, for use with in-circuit emulators and PROM programmers. *chex* produces the following formats:

- Motorola S-record format
- standard Intel hex format

clabs - absolute listing utility. *clabs* translates relocatable listings produced by the assembler by replacing all relocatable information by absolute information. This utility must to be used only after the linker.

clib - build and maintain object module libraries. *clib* allows you to collect related files into a single named library file for convenient storage. You use it to build and maintain object module libraries in standard library format.

cobj - object module inspector. *cobj* allows you to examine standard format executable and relocatable object files for symbol table information and to determine their size and configuration.

ct6805 - scan all the listing files, if any, and creates as output an assembly source file containing a replacement label followed by a jump instruction to the target function, for each of the selected function.

cv695 - IEEE695 format converter. *cv695* allows you to generate IEEE695 format file. This utility must to be used only after the linker.

cvdwarf - ELF/DWARF format converter. *cvdwarf* allows you to convert a file produced by the linker into an IELF/DWARF format file.

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 supports three programming models, allowing you to generate fully optimized code for your target system.

- The compiler uses registers **a** and **x** to hold the first argument of a function call if:

  1) the function does not return a structure and

  2) the first argument is derived from one of the following types:

  char,
  short,
  int,,
  pointer to...,
  or *array of....*

- 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:
- 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 compiler performs multiplication by powers of two as faster shift instructions.
- 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.

Support for ROMable Code

The compiler provides the following features to support ROMable code production. See Chapter 3 for more information.

- Referencing of absolute hardware addresses;
- Control of the MC68HC05 interrupt system;

- Automatic data initialization;
- User configurable runtime startup file;
- Support for mixing C and assembly language code; and
- User configurable executable images suitable for direct input to a PROM programmer or for direct downloading to a target system.

Support for eeprom

The compiler provides the following features to support **eeprom** handling:

- **@eeprom** type qualifier to describe a variable as an *eeprom* location. The compiler generates special sequences when the variable is modified.
- Library functions for erasure, initialization and copy of *eeprom* locations.

NOTE

The basic routine to program an eeprom byte is located in the library file ***eeprom.s*** *and has been written using the default input/output address* ***0x1000****. This file must be modified if using a different base address.*

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*.
For information on debugging support, see *Chapter 7*.
For information on using the programming utilities, see *Chapter 8*.
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 **acia.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.

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

Acia.c, Example file

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

```
/* EXAMPLE PROGRAM WITH INTERRUPT HANDLING
 */
#include <io.h>

#define SIZE      64          /* buffer size */
#define TDRE      0x80        /* transmit ready bit */

/*    Authorize interrupts. */
#define cli() _asm(“cli\n”)

/*    Some variables */
char buffer[SIZE];            /* reception buffer */
char * ptlec;                 /* read pointer */
char * ptecr;                 /* write pointer */

/*    Character reception.
 *    Loops until a character is received.
 */
char getch(void)
      {
      char c;                 /* character to be returned */

      while (ptlec == ptecr)  /* equal pointers => loop */
            ;
      c = *ptlec++;           /* get the received char */
      if (ptlec >= &buffer[SIZE])/* put in in buffer */
            ptlec = buffer;
      return (c);
      }

/*    Send a char to the SCI.
 */
void outch(char c)
      {
      while (!(SCSR & TDRE))  /* wait for READY */
            ;
      SCDR = c;               /* send it */
      }

/*    Character reception routine.
 *    This routine is called on interrupt.
 *    It puts the received char in the buffer.
 */
@interrupt void recept(void)
      {
      *ptecr = SCSR;               /* clear interrupt */
      *ptecr++ = SCDR;             /* get the char */
      if (ptecr >= &buffer[SIZE])  /* put it in buffer */
            ptecr = buffer;
      }

/*    Main program.
 *    Sets up the SCI and starts an infinite loop
 *    of receive transmit.
 */
void main(void)
      {
      ptecr = ptlec = buffer; /* initialize pointers */
      BAUD = 0x30;            /* initialize SCI */
      SCCR2 = 0x2c;           /* parameters for interrupt */
      cli();                  /* authorize interrupts */
      for (;;)                /* loop */
            outch(getch());   /* get and put a char */
      }
```

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, **cx6805** 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.h05	input (*e.g.* libraries) or output (*e.g.* absolute executable) file for the linker

Compiling and Linking

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

```
cx6805 acia.c
```

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

```
acia.c:
```

The result of the compilation process is an object module named *acia.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 **acia.s**.

```
cx6805 -s acia.c
```

The **-s** option directs **cx6805** to stop after having produced the assembly file *acia.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 acia.s
```

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

```
cx6805 -l acia.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:

```
cx6805 -v -s acia.c
```

the display will look like something similar to the following:

```
acia.c:
      cp6805 -o \2.cx1 -i\cx\h6805 -u acia.c
      cg6805 -o \2.cx2 \2.cx1
      co6805 -o acia.s \2.cx2
```

The compiler runs each pass:

cp6805	the C parser
cg6805	the assembly code generator
co6805	the optimizer

Step 2: Assembler

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

```
cx6805 acia.s
```

or

```
ca6805 -i\cx\h6805 acia.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 *acia.ls* that will also contain a cross reference:

```
ca6805 -c -l acia.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**) into an absolute executable file: **acia.h05** in our example. Code and data sections will be located at absolute memory addresses. The linker is used with a command file (*acia.lkf* in this example).

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

Type	Description
.text	code (or program) section (*e.g.* **ROM**)
.const	constant and literal data (*e.g.* **ROM**)
.data	initialized data in external memory (see **@near** in chapter 3)
.bss	non initialized data in external memory
.bsct	initialized data in the first 256 bytes (see **@tiny** in chapter 3), also called **zero page**
.ubsct	non initialized data in the *zero page*

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

```
line 1 # LINK COMMAND FILE FOR TEST PROGRAM
line 2 # Copyright (c) 1995 by COSMIC Software
line 3 #
line 4 +seg .text -b 0x3000   # program start address
line 5 +seg .bsct -b 0x20    # zero page start address
line 6 +seg .data -b 0x100   # data start address
line 7 crts.o               # startup routine
line 8 acia.o               # application program
line 9 \cx\lib\libi.h05     # C library (if needed)
line 10 \cx\lib\libm.h05    # machine library
line 11 +seg .text -b0x3ff8 # vectors start address
line 12 vector.o            # interrupt vectors file
line 13 +def __memory=@.bss # symbol used by startup
```

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

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

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

line 4: **`+seg .text -b0x3000`** creates a text (code) segment located at **`3000`** (hex address)

line 5: **`+seg .bsct -b0x20`** creates a zero page segment located at **`20`** (hex address)

line 6: **`+seg .data -b0x100`** creates a data segment located at **`100`** (hex address)

line 7: **`crts.o`** runtime startup code. It will be located at **`0x3000`** (code segment)

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

line 9: **`libi.h05`** the integer library to resolve references

line 10: **`libm.h05`** the machine library to resolve references

line 11: **`+seg .text -b0x3ff8`** creates a new segment text (code) segment located at **`3ff8`** (hex address)

line 12: **`vectors.o`** interrupt vectors file

line 13: **`+def __memory=@.bss`** defines a symbol **__memory** equal to the value of the current address in the **.bss** segment. This is used to get the address of the end of the *bss*. The symbol *__memory* is used by the startup routine to reset the *bss*.

By default, and in our example, the *.bss* segment follows the *.data* segment.

The *crts.o* file contains the runtime startup that performs the following operations:

- initialize the *bss*, if any
- initialize the stack pointer
- call *main()* or any other chosen entry point.

For more information, see "*Modifying the Runtime Startup*" in Chapter 3.

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

```
clnk -o acia.h05 acia.lkf
```

Step 4: Generating S-Records file

Although *acia.h05* is an executable image, it may not be in the correct format to be loaded on your target. Use the **chex** utility to translate the format produced by the linker into standard formats. To translate *acia.h05* to *Motorola standard S-record* format:

```
chex acia.h05 > acia.hex
```

or

```
chex -o acia.hex acia.h05
```

acia.hex is now an executable image in *Motorola S-record* format and is ready to be loaded in your target system.

For more information, see "*The chex Utility*" in Chapter 8.

Linking Your Application

You can create as many *text*, *data* and *bss* segments as your application requires. For example, assume we have one *bss*, two *data* and two *text* segments. Our link command file will look like:

```
+seg .bsct -b0x20             # zpage start address
var_zpage.o                   # file with zpage variable
+seg .text -b 0x1000 -n .text # program start address
```

```
+seg .const -a .text            # constant follow
+seg .data -b 0x100             # data start address
+seg .bss -b 0x200              # bss start address
crts.o                          # startup routine
acia.o                          # main program
module1.o                       # application program
+seg .text -b 0x2000            # start new text section
module2.o                       # application program
module3.o                       # application program
\cx\lib\libi.h05                # C library (if needed)
\cx\lib\libm.h05                # machine library
+seg .text -b0x3ff8             # vectors start address
vector.o                        # interrupt vectors
+def __memory=@.bss             # symbol used by startup
```

In this example the linker will locate and merge *crts.o*, *acia.o* and *module1.o* in a *text* segment at **`0x1000`**, a *data* segment at **`0x100`** and a *bss* segment, if needed at **`0x200`**. *zero page* variables will be located at **`0x20.`** The rest of the application, *module2.o* and *module3.o* and the libraries will be located and merged in a new *text* segment at **`0x2000`** then the interrupt vectors file, *vector.o* in a *text* segment at **`0x3ff8`**. All constants will be located after the first *text* segment.

For more information about the linker, see Chapter 6, "*Using The Linker*".

Optimize Function Call

The compiler can optimize the function call, which produce less code but creating a time overhead at each function call. You must, first, specify the **-l** option to **cx6805** to generate listing file, then link the full application, call the **ct6805** utility to produce the jump table, **jmptab.s**, and rebuild the full application by specifying the **+jmp** option to **cx6805**. From the example provided with the package, type the following commands:

```
cx6805 -vl acia.c vector.c
clnk -o acia.h05 acia.lkf
ct6805 -o jmptab.s acia.h05
cx6805 -vl +jmp acia.c vector.c
clnk -o acia.h05 acia.lkf
```

For more information, see "*Function Call Optimization*" in Chapter 3.

Generating Automatic Data Initialization

Usually, in embedded applications, your program must reside in ROM.

This is not an issue when your application contains code and read-only data (such as string or const variables). All you have to do is burn a PROM with the correct values and plug it into your application board.

The problem comes up when your application uses initial data values that you have defined with initialized static data. These static data values must reside in RAM.

There are two types of static data initializations:

1) data that is explicitly initialized to a non-zero value:

```
char var1 = 25;
```

which is generated into the **.bsct** section and

2) data that is explicitly initialized to zero or left uninitialized:

```
char var2;
```

which is generated into the **.ubsct** section.

There is one exception to the above rules when you declare data that will be located in the **external memory**, using the **@near** type qualifier. In this case, the data is generated into the **.data** section if it is initialized or in the **.bss** section otherwise.

The first method to ensure that these values are correct consists in adding code in your application that reinitializes them from a copy that you have created and located in ROM, at each restart of the application.

The second method is to use the **crtsi.h05** (or **crtsx.h05** for variables located in *external memory*) start-up file:

1) that defines a symbol that will force the linker to create a copy of the initialized RAM in ROM

2) and that will do the copy from ROM to RAM

The following link file demonstrates how to achieve automatic data initialization.

```
# demo.lkf: link command with automatic init
+seg .text -b 0x1000 -n .text       # program start address
+seg .const -a .text                # constant follow
+seg .bsct -b 0x20 -n iram -m 0x100 # zpage start address
+seg .share -a iram -is             # shared segment
+seg .data -b0x100                  # data start address
\cx\lib\crtsi.h05                   # startup with auto-init
acia.o                              # main program
module1.o                           # module program
\cx\lib\libi.h05                    # C library (if needed)
\cx\lib\libm.h05                    # machine library
+seg .text -b 0x3ff8                # vectors start address
vector.o                            # interrupt vectors
+def __memory=@.bss                 # symbol used by library
```

In the above example, the *text* segment is located at address **`0x1000`**, the *data* segment is located at address **`0x100`**, immediately followed by the *bss* segment that contains uninitialized data. The initialized data in ROM will follow the descriptor created by the linker after the code segment.

In case of multiple code and data segments, a link command file could be:

```
# demoinit.lkf: link command with automatic init
+seg .text -b 0x1000 -n .text     # program start address
+seg .const -a .text              # constant follow
+seg .bsct -b 0x20 -n iram -m 0x100# zpage start address
+seg .share -a iram -is           # shared segment
+seg .data -b0x100                # data start address
\cx\lib\crtsi.h05                 # startup with auto-init
acia.o                            # main program
module1.o                         # module program
+seg .text -b0x1800               # new code segment
module2.o                         # module program
module3.o                         # module program
\cx\lib\libi.h05                  # C library (if needed)
\cx\lib\libm.h05                  # machine library
+seg .text -b 0x3ff8              # vectors start address
vector.o                          # interrupt vectors
+def __memory=@.bss               # symbol used by startup
```

or

```
# demoinit.lkf: link command with automatic init
+seg .text -b 0x1000 -n .text    # program start address
+seg .const -a .text             # constant follow
+seg .bsct -b 0x20 -n iram -m 0x100# zpage start address
+seg .share -a iram -is          # shared segment
+seg .data -b0x100               # data start address
\cx\lib\crtsi.h05                # startup with auto-init
acia.o                           # main program
module1.o                        # module program
+seg .text -b0x1800 -it          # set the section attribut
module2.o                        # module program
module3.o                        # module program
\cx\lib\libi.h05                 # C library (if needed)
\cx\lib\libm.h05                 # machine library
+seg .text -b 0x3ff8             # vectors start address
vector.o                         # interrupt vectors
+def __memory=@.bss              # symbol used by startup
```

In the first case, the initialized data will be located after the first code segment. In the second case, the **-it** option instructs the linker to locate the initialized data after the segment marked with this flag. The initialized data will be located after the second code segment located at address **`0x1800`**.

For more information, see “*Initializing data in RAM*” in Chapter 3 and “*Automatic Data Initialization*” in Chapter 6.

Specifying Command Line Options

You specify command line options to **cx6805** to control the compilation process.

To compile and get a relocatable file, type:

```
cx6805 acia.c
```

The file produced is *acia.o*.

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 *acia.ls*.

To perform the operations described above, enter the command:

```
cx6805 -v -l acia.c
```

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

- the C source file **acia.c**
- the C and assembly language listing **acia.ls**
- the object module **acia.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:

```
cx6805 -cl\mylist -co\myobj -l acia.c
```

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

cx6805 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:

```
cx6805 -v start.s acia.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 MC68HC05 applications.

Introduction

This chapter provides details about:

Modifying the Runtime Startup

The runtime startup module performs many important functions to establish a runtime environment for C. The runtime startup file included with the standard distribution provides the following:

- Initialization of the **bss** section if any,
- ROM into RAM copy if required,
- Initialization of the stack pointer,
- *_main* or other program entry point call, and
- An exit sequence to return from the C environment. Most users must modify the exit sequence provided to meet the needs of their specific execution environment.

The following is a listing of the standard runtime startup file **crts.h05** included on your distribution media. It does not perform automatic data initialization. A special startup program is provided, **crtsi.h05**, which is used instead of *crts.h05* when you need automatic data initialization (or **crtsx.h05** for data located in external memory). The runtime startup file can be placed anywhere in memory. Usually, the startup will be "linked" with the **RESET** interrupt, and the startup file may be at any convenient location.

```
;  C STARTUP FOR MC68HC05
;  Copyright (c) 1995 by COSMIC Software
;
   xref  _main
   xdef  _exit, __stext, c_h
;
   switch.text
__stext:
   lda   #$81        ; RTS
   sta   c_h+2       ; opcode in place
   rsp               ; reset stack pointer
   jsr   _main       ; execute main
_exit:
   bra   _exit       ; and stay here
;
;  area for external memory access
```

```
17 ;
18    switch.ubsct
19    ds.b  1          ; opcode
20 c_h:
21    ds.b  3          ; MSB + LSB + rts
22 ;
23    end
```

Description of Runtime Startup Code

_main is the entry point into the user C program.

Line 11 resets the stack pointer.

Line 12 calls *main()* in the user's C program.

Lines 13 to 14 trap a return from *main()*. If your application must return to a monitor, for example, you must modify this line.

Lines 18 to 21 reserve bytes for external memory access

Initializing data in RAM

If you have initialized static variables, which are located in **RAM**, you need to perform their initialization before you start your C program. The **clnk** linker will take care of that: it moves the initialized data segments after the **first** text segment, or the one you have selected with the **-it** option, and creates a descriptor giving the starting address, destination and size of each segment.

The table thus created and the copy of the **RAM** are located in **ROM** by the linker, and used to do the initialization. An example of how to do this is provided in the **crtsi.s** (or **crtsx.s** for variables located in external memory) file, located in the headers sub-directory.

```
;     C STARTUP FOR MC68HC05
;     WITH AUTOMATIC DATA INITIALISATION
;     Copyright (c) 1995 by COSMIC Software
;
      xref  _main, __memory, __idesc__
      xdef  _exit, __stext, c_h, c_reg
;
      switch.text
__stext:
      rsp                  ; reset stack pointer
      lda   #$81           ; RTS
      sta   c_h+2          ; opcode in place
      lda   #$D6           ; LDA IX2
      sta   c_h-1          ; opcode in place
      clrx                 ; start index
ibcl:
      lda   __idesc__+2,x ; test flag byte
      beq   prog           ; no more segment
      lda   __idesc__+1,x ; compute start
      sub   __idesc__+4,x ; offset by ram address
      sta   c_h+1          ; in read vector
      lda   __idesc__,x    ; because sharing
      sbc   #0             ; the same
      sta   c_h            ; index
      lda   __idesc__+6,x ; compute length
      sub   __idesc__+1,x ; of segment
      add   __idesc__+4,x ; ram end address
      sta   c_reg          ; save for compare
      stx   c_reg+1        ; save index
      ldx   __idesc__+4,x ; load ram address
dbcl:
      jsr   c_h-1          ; load byte
      sta   0,x            ; store it
      incx                 ; next byte
      cpx   c_reg          ; end address
      bne   dbcl           ; no, loop back
      lda   c_reg+1        ; get back index
      add   #5             ; next descriptor
      tax                  ; in place
      bra   ibcl           ; and loop
prog:
      jsr   _main          ; execute main
_exit:
      bra   _exit          ; and stay here
;
;     area for external memory access
;
      switch.ubsct
      ds.b  1              ; opcode
c_h:
      ds.b  3              ; MSB + LSB + rts
c_reg:
      ds.b  2              ; extra accumulator
;
      end
```

crtsi.s performs the same function as described with the *crts.s*, but with one additional step. Lines (marked in bold) in *crtsi.s* include code to copy the contents of initialized static data, which has been placed in the text section by the linker, to the desired location in RAM.

For more information, see "*Generating Automatic Data Initialization*" in Chapter 2, "*Tutorial Introduction*" and "*Automatic Data Initialization*" in Chapter 6, "*Using The Linker*".

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 char 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.

The *const* keyword implies the **@near** memory space to allow such a variable to be located in the code space. If a memory space modifier is explicitly given on a declaration using the *const* keyword, the compiler uses the given space instead of the default one, meaning that the object may not be located in the code space depending on the memory space given. In such a case, the *const* keyword still enforces the assignment checking.

You may specify *const* and *volatile* together, in either order. A *const volatile* data object could be a Read-only status register, or a status 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 y;   /* volatile pointer to int */
const  float  pi  =  355.0  /  113.0;  /*  pi  is  never
```

Performing Input/Output in C

You perform input and output in C by using the C library functions *getchar, gets, printf, putchar, puts* and *sprintf*. 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);
```

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 usefull 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 **`0x0e`** with the identifier name *SCCR1*, write a definition of the form:

```
char SCCR1 @0x0e;
```

where **`@0x0e`** indicates an absolute address specification and not a data initializer. Since input/output on the MC68HC05 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 = SCCR1; /* to read from input port */
SCCR1 = c; /* to write to output port */
```

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

```
#define SCCR1 *(char *)0x0e
```

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.

Another example of how to reference a direct memory address, defines a structure at absolute address **`0x6000`**:

```
struct acia
{
char status;
char data;
} acia @0x6000
```

Using this declaration, references to **`acia.status`** will refer to memory location **`0x6000`** and **`acia.data`** will refer to memory location **`0x6001`**. This is very useful if you are building your own custom I/O hardware that must reside at some location in the 68HC05 memory map.

Accessing Internal Registers

All registers are declared in the **io.h** file provided with the compiler. This file should be included in each file using the input-output registers, for example by a:

```
#include <io.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. All those definitions are already provided in the **io.s** file which may be included in an assembly source by a:

```
        include "io.s"
```

Note that the compiler will access to these registers as standard variables. In some case of reading or writing some "*int*" registers, you should declare an union (with two char and one int) instead of using directly the I/O register.

Placing Data Objects in The Bss Section

The compiler automatically reserves space for data objects initialized to zero or uninitialized data object. All such data are placed in the **.bss** section. All initialized static data are placed in the **.data** section. The bss section is located, by default, after the data section by the linker.

The run-time startup codes, *crts.s* and *crtsi.s*, contain code which initializes the bss section space reservations to zero.

The compiler provides a special option, **+nobss**, which forces the compiler to put data initialized to zero or uninitialized data to be explicitly placed in the **.data** section.

Placing Data Objects in Internal Memory

The compiler allocates all the variables in the **zero page** by default. Such variables will be located into the section **.bsct** if they are initialized, or in the section **.ubsct** otherwise. An external object name is published via a **xref.b** declaration at the assembly language level. A variable can be explicitely allocated in *zero page* by using the **@tiny** modifier:

```
@tiny char c;
```

NOTE

The code generator does not check for zero page overflow.

Setting Zero Page Size

You can set the size of the *zero page* section of your object image at link time by specifying the following options on the linker command line:

```
+seg .bsct -m##
```

where ## represents the size of the *zero page* section in bytes. Note that the size of the zero page section can never exceed **256 bytes**.

Placing Data Objects in External Memory

The compiler allows variables to be allocated in **external memory** by using the **@near** modifier. Such variables will be located into the **.data** section if they are initialized, or in the **.bss** section otherwise. An external object name is published via a **xref** declaration at the assembly language level. The following declaration:

```
@near char ext;
```

instructs the compiler to locate the variable *ext* in the external memory.

To place data objects into *external memory* on a file basis, you use the **#pragma** directive of the compiler. The compiler directive:

```
#pragma space [] @near
```

instructs the compiler to place all data objects of storage class **extern** or **static** into external memory for the current unit of compilation (usually a file).

The section must end with a **#pragma space [] @ tiny** to revert to the default compiler behaviour.

Placing Data Objects in the EEPROM Space

The compiler allows the use to define a variable as an **eeprom** location, using the type qualifier **@eeprom**. This causes the compiler to produce special code when such a variable is modified. When the compiler detects a write to an *eeprom* location, it calls a machine library function which performs the actual write. An example of such a definition is:

```
@eeprom char var;
```

To place all data objects from a file into *eeprom*, you can use the **#pragma** directive of the compiler. The directive

```
 #pragma space [] @eeprom @near
```

instructs the compiler to treat all *extern* and *static* data in the current file as *eeprom* locations. The **@near** modifier is necessary because the eeprom is located outside the zero page.

The section must end with a **#pragma space [] @near** or **@tiny**, depending on the memory model selected.

The compiler allocates *@eeprom* variables in a separate section named **.eeprom**, which will be located at link time. The linker directive:

```
 +seg .eeprom -b0xb600 -m512
 var_eeprom.o
```

will create a segment located at address **0xb600**, with a maximum size of 512 bytes.

NOTE

The code generator cannot check if data address will be eeprom addresses after linkage.

Redefining Sections

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

Section	Description
.text	executable code
.const	text string and constants
.data	initialized variables (**@near**)
.bss	uninitialized variables (**@near**)
.bsct	initialized variables in zero page (**@tiny** by default)
.ubsct	uninitialized variables in zero page (**@tiny** by default)
.eeprom	any variable in eeprom (**@eeprom**)

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 one of the following sequences:

const
@tiny
@near
@eeprom

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 *bss* section forces the compiler to produce the memory definitions for all the previous *bss* declarations before to switch to the new section.

The following directives:

```
#pragma section (code)
#pragma section const {string}
#pragma section @near [udata]
#pragma section @near {idata}
#pragma section [uzpage]
#pragma section {izpage}
#pragma section @eeprom @near {e2prom}
```

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

- executable code is redirected to section **.code**
- strings and constants are redirected to section **.string**
- uninitialized variables are redirected to section **.udata**
- initialized data are redirected to section **.idata**
- uninitialized zpage variables are redirected to section **.uzpage**
- initialized zpage variables are redirected to section **.izpage**
- eeprom variables are redirected to section **.e2prom**

Note that **{name}** and **[name]** are equivalent for constant, zero page and eeprom sections as they are all considered as initialized.

The following directive:

```
#pragma section ()
```

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

Local Variables and Arguments

The compiler does not use any stack as the processor itself does not provide such a feature, and allocates local variables and arguments in internal memory at fixed addresses. This implies that this compiler cannot build recursive functions. By default, the compiler instructs the linker to optimize the memory allocation of those areas, by sharing the memory areas corresponding to functions never calling each other. This mechanism can be defeated by specifying the **+nsh** compiler option, then allocating a separate area for each function, or by using the type qualifier **@noshare** on a specific function, then allocating a separate area for that function only.

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 connexion 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 { */
sec; set carry bit
rol asmvar; access assembler variable
#endasm
else
test = 1;
}
```

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:

```
  rsp
  jsr _main
```

you would write:

```
  _asm("rsp\n jsr _main\n");
```

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

_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:

```
  var = _asm("asra\n rola\n rola\n", var);
```

allows to rotate the variable *var* passed as argument in the **a** register, and store the result in the same variable. The variable *var* is supposed to be declared as a *char*, and is loaded in the **a** register because it is considered as a first argument. The result is expected in the **a** register in order to comply with the return register convention, as described below.

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.*

Writing Interrupt Handlers

A function declared with the type qualifier **@interrupt** is suitable for direct connection to an interrupt (hardware or software). *@interrupt* functions may not return a value. *@interrupt* functions are allowed to have arguments, although hardware generated interrupts are not likely to supply anything meaningful.

IMPORTANT

A function cannot be called by an interrupt function and by a standard function, because the function is then included in more than one graph. Otherwise the linker will output an error message (function is reentrant).

When you define an *@interrupt* function, the compiler uses the “**rti**” instruction for the return sequence, and saves, *if necessary*, the memory bytes used by the compiler for its internal usage. Those areas are **c_reg** (up to 2 bytes), **c_h** (3 bytes) and **c_lreg** (4 bytes). Those bytes will be saved and restored if the interrupt function uses them directly. If the interrupt function does not uses these areas directly, but calls another C function, the *c_reg* and *c_h* areas will be automatically saved and restored, unless using the type qualifier **@nosvf** on the interrupt function definition. This qualifier can be used when the called functions are known not using those areas, but the compiler does not perform any verification. The *c_lreg* area is not saved implicitely in such a case, in order to keep the interrupt function as efficient as possible. If any function called by the interrupt function uses **longs** or **floats**, the *c_lreg* area can be saved by using the type qualifier **@svlreg** on the interrupt function definition. Extra bytes will be added to the local variables of the interrupt function to hold the copy.

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)
{
...
}
```

You cannot call an *@interrupt* function directly from a C function. It must be connected with the interrupt vectors table.

NOTE

The @interrupt modifier is an extension to the ANSI standard.

Placing Addresses in Interrupt 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 .text
xref handler1, handler2, handler3
vector1:dc.w handler1
vector2:dc.w handler2
vector3:dc.w handler3
end
```

where *handler1* and so forth are interrupt handlers.

A small C routine that performs the same operation is:

```
extern void handler1(), handler2(), handler3();
void (* const vector[])() =
{
handler1,
handler2,
handler3,
};
```

where *handler1* and so forth are interrupt handlers. Then, at link time, include the following options on the link line:

```
+seg .const -b0x3ff8 vector.o
```

where *vector.o* is the file which contains the vector table. This file is provided in the compiler package.

Function Call Optimization

Some 68HC05 derivatives contain a small ROM area in the *zero page* addressing range. This small area may be used as a jump table containing jump instructions to the mostly used functions of the application. These functions can be accessed through this jump table using the *direct* addressing mode instead of the *extended* addressing mode, thus saving one byte for each function call, but creating a time overhead at each function call.

The compiler is able to implement this feature in an almost automatic way, by using the following sequences of operations:

1) build first the full application while compiling all the files with the **-l** option to create assembly listings.

2) Run the **ct6805** utility to create the jump table from the result of the linker.

3) Rebuild the full application while compiling all the files with the **+jmp** option, allowing the optimizer to perform automatically the function name replacement.

NOTE

The current implementation requires the jump table file name to be: ***jmptab.s***.

The *ct6805* utility reads the executable file produced by the linker and finds the name of all the object files of the application. *ct6805* then scans all the listing files, if any, and creates as output an assembly source file containing a replacement label followed by a jump instruction to the target function, for each of the selected function. The selected function list is built by extracting the sixteen most used function names following a *jsr* instruction, including the library functions.

The jump table file must be assembled and linked at the appropriate address.

For example, from the *acia.h05* file built by the *test* program provided with the package, run the following command:

```
ct6805 -o jmptab.s acia.h05
```

This command produces the following result in the *jmptab.s* file:

```
;      JUMP TABLE FOR 68HC05
;      Copyright (c) 1995 by COSMIC Software
;
R_main:
       jmp   _main
R_outch:
       jmp   _outch
R_getch:
       jmp   _getch
;
       xdef  R_main
       xref  _main
       xdef  R_outch
       xref  _outch
       xdef  R_getch
       xref  _getch
       end
```

Then, compiling the *acia.c* with the following command:

```
cx6805 -vl +jmp acia.c
```

will generate (extract from the *acia.ls* listing file):

```
...
79   0045 bd00      jsrR_getch
81   0047 bd00      jsrR_outch
```

```
84   0049 20fa       braL32
85                   xref.bR_outch
86                   xref.bR_getch
87                   xdef_main
88                   xdef_recept
...
```

NOTE

*The function names which have been replaced are prefixed with **R**.*

Builtin Function

The compiler allows access to specific instructions or features of the 68HC05 processor, using **@builtin** functions. Such functions shall be declared as external functions with the *@builtin* modifier. The compiler recognizes three predefined functions when explicitly declared as follows:

```
@builtin carry(void);
@builtin irq(void);
@builtin imask(void);
```

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 the **a** register to 0 or 1 depending on the carry bit value.
irq	the *irq* function is used to test the interrupt line level using the **bih** or **bil** instruction. The *irq* function can be used only in a test
imask	the *imask* function is used to test the interrupt mask bit in the condition register using the **bms** or **bmc** instruction. The *imask* function can be used only in a test.

A full description with examples is provided in Chapter 4.

Any other function declared as an *@builtin* will be translated into a call to a user provided macro. The macro name is obtained by prefixing the

@builtin function name with the '_' character. Arguments are allowed but should be restricted to variable references. Each reference is translated into the proper assembler expression (same translation as applied by the compiler) and then passed to the macro as a quoted text string.

@builtin functions may use the registers **a** and/or **x**, but the compiler can not check their use and will not save them. To save the registers before they are used by *@builtin* functions, you must add the **@usea** and/or the **@usex** modifiers. For example:

```
  @builtin @usea lsub();
```

tells the compiler that *lsub()* uses the register **a**, so that the compiler will save it. If both registers are used, you must specify both modifiers.

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 **.text** 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 **.const** section. An option on the code generator allows such constants to be produced in the **.text** section

The compiler generates initialized data declared with the **@near** modifier into the **.data** 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 **.const** section. Initialized data declared with the **@tiny** space modifier will be generated into the **.bsct** section. Such external data names are published via **xref.b** declarations. Uninitialized data are normally generated into the **.bss** section for **@near** variables or the **.ubsct** section for **@tiny** variables, unless forced to the **.data** or **.bsct** section by the compiler option **+nobss**.

.bsct	@tiny char i =2;	xdef
.ubsct	@tiny char i;	xdef
.data	int init = 1	xdef
.bss	int uninit	xdef
.text	char putchar(c);	xdef
Any of above	extern int out;	xref

Function calls are performed according to the following:

1) Arguments are evaluated from right to left. The first argument is stored in the **a** register if it is a char, or in the **a, x** register pair if its type is *short* or *int*, and if the function does not return a structure.

2) The function is called via a *jsr _func* instruction.

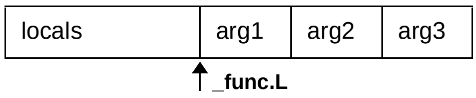
Register Usage

Except for the return value, the registers **a**, **x** and the condition codes are undefined on return from a function call. The return value is in **a** if it is of type char or pointer to internal ram, **a, x** if it is of type short, integer or pointer to external ram or code space (*const*). The return value is in the memory located at symbol **c_lreg** if it is of type long or float.

In order to access local variables and arguments, the compiler creates a memory area and a symbol used to access this area. The symbol name is obtained by adding the **.L** suffix to the function name. This symbol is made public if the function is not declared with the *static* C keyword. The first argument may be hold in register, and will be stored at the function entry. Such a function declaration:

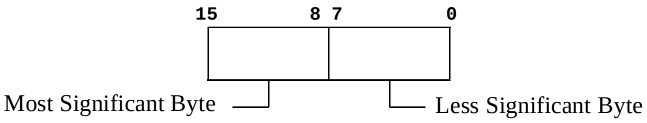
```
int func(int arg1, int arg2, int arg3)
```

will create the following memory area:



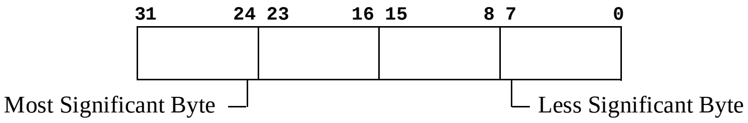
Data Representation

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



Short Int

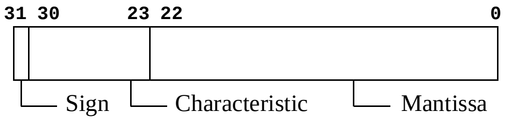
Data objects of type *long integer* are stored as four bytes, in descending order of significance.



Long

Plain *pointers* are stored as one byte. *@near* pointers (external or code memory) are stored as two bytes.

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

To invoke the cross compiler, type the command **cx6805**, 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:

```
cx6805 [options] <files>.[c|s]
```

cx6805 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, **cx6805** 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:

```
cx6805 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, **cx6805** 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 **cx6805** compiler accepts the following command line options, each of which is described in detail below:

```
  cx6805 [options] <files>
         -a*>  assembler options
         -ce*  path for errors
         -cl*  path for listings
         -co*  path for objects
         -d*>  define symbol
         -e    create error file
         -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 directoy than the source files.

-cl*	specify a path for the listing files. By default, listings are created in the same directoy than the source files.
-co*	specify a path for the object files. By default, objects are created in the same directoy 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 20 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 if there are errors.
-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 behavior but reduces the command line lenght.
-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 *cx6805.cxf* in the compiler directory as specified in the installation process.
-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.
+jmp	optimize function calls. This option uses a jump table generated by *ct6805* which contains the jump function call to be used. For more information, see "*Function Call Optimization*" in Chapter 3.

+nobss do not use the *.bss* section for variables allocated in external memory. By default, such 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 contants in the code section **.text** instead of the specific section **.const**.

+nsh do not share memory areas allocated for local variables and arguments. By default, the linker optimizes the memory allocation by sharing the memory areas corresponding to independent functions.

+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*.

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
cp6805	<file>.c	<file>.1
cg6805	<file>.1	<file>.2
co6805	<file>.2	<file>.s
error listing	<file>.c	<file>.err
assembler listing	<file>.[c|s]	<file>.ls
C header files	<file>.h	
ca6805	<file>.s	<file>.o
source listing	<file>.s	<file>.ls
clnk	<file>.o	name required
chex	<file>	STDOUT
clabs	<file.h12>	<files>.la
clib	<file>	name required
cobj	<file>	STDOUT
ct6805	<file.h05>	jmptab.s

Generating Listings

You can generate listings of the output of any (or all) the compiler passes by specifying the **-l** option to **cx6805**. 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:

```
cx6805 -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 **cx6805**. You can locate the error file in a different directory by using the **-ce** option. For example, you would type:

```
cx6805 -e prog.c
```

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

Generating Jump Table

You can generate a jump table, to optimize the function call, by specifying, first, the command line option **-l** to the **cx6805** compiler, then link the full application, call the *ct6805* utility to produce the jump table, *jmptab.s*, and rebuild the full application by specifying the **+jmp** option to the **cx6805** compiler. For more information, see “*Optimize Function Call*” in Chapter 2, “*Tutorial Introduction*”, and “*Function Call Optimization*” in Chapter 3, “*Programming Environments*”.

Return Status

cx6805 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:

```
cx6805 -v file.c
```

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

```
cx6805 -s file.c
```

C Library Support

This section describes the facilities provided by the C library. The C cross compiler for MC68HC05 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.

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     isupper   strcmp   xmemmove
abs      isxdigit  strcpy   xmemset
atoi     labs      strcspn  xputs
atol     ldiv      strlen   xstrcat
div      memchr    strncat  xstrchr
```

eepera	memcmp	strncmp	xstrcmp
getchar	memcpy	strncpy	xstrcpy
gets	memmove	strpbrk	xstrcspn
isalnum	memset	strrchr	xstrlen
isalpha	printf	strspn	xstrncat
iscntrl	putchar	strstr	xstrncmp
isdigit	puts	strtol	xstrncpy
isgraph	rand	tolower	xstrpbrk
islower	sprintf	toupper	xstrrchr
isprint	srand	xmemchr	xstrspn
ispunct	strcat	xmemcmp	xstrstr
isspace	strchr	xmemcpy	xstrtol

Floating Point Library Functions

acos	cosh	log	sprintf
asin	exp	log10	sqrt
atan	fabs	modf	strtod
atan2	floor	pow	tan
atof	fmod	printf	tanh
ceil	frexp	sin	
cos	ldexp	sinh	

Common Input/Output Functions

Two of the functions that perform stream 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*, *sprintf*.

Functions Implemented as Macros

Two 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* and *min*.

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, iscntrl, isgraph, isprint, ispunct, isspace, isxdigit, isdigit, isupper, islower, tolower* and *toupper*.

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

<io.h> - Header file for input-output registers. Each register has an upper-case name which matches the standard Motorola definition. Note that because there is a wide range of derivatives, this file should be used as a template to derive an accurate file for a given processor.

<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*.

<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* and *sprintf*.

<stdlib.h> - Header file for general utilities: *abs*, *abort*, *atof*, *atoi*, *atol*, *div*, *exit*, *labs*, *ldiv*, *rand*, *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 are: *isalnum, isalpha, iscntrl, isgraph, isprint, ispunct, isspace, isxdigit, isdigit, isupper, islower, sbreak, tolower* and *toupper.*

Usage of External Memory Pointers

Most of the library functions expect pointers into **internal memory** meaning that any pointer arguments is stored on one byte. The *printf* and *sprintf* functions are prototyped to receive the format string as a **@near** pointers and should display strings from external memory using the uppercase **S** format character. All other functions using pointers are duplicated using a derived name built by prefixing an original with the '**x**' letter. Such functions receive and return if necessary, **@near** pointers. For more information, see "*Placing Data Objects in External Memory*" in Chapter 3.

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 *tsx; jsr _main*, may be generated by the following call:

```
_asm(“\ttsx\n\tjsr _main\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.

CHAPTER

5

Using The Assembler

The **ca6805** cross assembler translates your assembly language source files into relocatable object files. The C cross compiler calls *ca6805* to assemble your code automatically, unless specified otherwise. *ca6805* generates also listings if requested. This chapter includes the following sections:

Invoking ca6805

ca6805 accepts the following command line options, each of which is described in detail below:

```
ca6805 [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 listing
        +l*   listing file name
        -m    accept old syntax
        -mi   accept label syntax
        -o*   output file name
        -pe   all equates public
        -p    all symbols public
        -pl   keep local symbol
        -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, unless the **+l** option has been specified.
+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*.

-p	mark all defined symbols as **public**. This option has the same effect than adding a **xdef** directive for each label.
-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.
-u	produce an error message in the listing file for all occurence 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, *ca6805* generates an object file. If requested by the **-l** or **-c** options, *ca6805* 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 *ca6805* 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

ca6805 recognizes the following instructions:

adc	bhi	brset	decx	mul	sbc
add	bhs	bset	eor	neg	sec
and	bih	bsr	inc	nega	sei
asl	bil	clc	inca	negx	sta
asla	bit	cli	incx	nop	stop
aslx	blo	clr	jmp	ora	stx
asr	bls	clra	jsr	rol	sub
asra	bmc	clrx	lda	rola	swi
asrx	bmi	cmp	ldx	rolx	tax
bcc	bms	com	lsl	ror	tst
bclr	bne	coma	lsla	rora	tsta
bcs	bpl	comx	lslx	rorx	tstx
beq	bra	cpx	lsr	rsp	txa
bhcc	brclr	dec	lsra	rti	wait
bhcs	brn	deca	lsrx	rts	

The **operand** field of an instruction uses an addressing mode to describe the instruction argument. The following example demonstrates the accepted syntax:

```
tax                    ; implicit
lda     #1             ; immediate
and     var            ; direct or extended
add     ,x             ; indexed
or      0,x            ; indexed
bne     loop           ; relative
bset    1,var          ; bit number
brset   2,var,loop     ; bit test and branch
```

The assembler chooses the smallest addressing mode where several solutions are possible. *Direct* addressing mode is selected when using a label defined in the *.bsct* section.

For an exact description of the above instructions, refer to the Motorola's *M68HC05 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 '*a*' and '*x*' 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.

```
wait:    macro
\@loop:  brset       1,PORTA,\@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$:     deca
        bne     1$
2$:     decb
        bne     2$
```

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$:    lda     var
        beq     10$
        sta     var2
        local
10$:    lda     var2
        beq     10$
        sta     var
        rts
function2:
10$:    lda     var2
        sub     var
        bne     10$
        rts
```

Constants

The assembler accepts **numeric** constants and **string** constants. *Numeric* constants are expressed in different bases depending on a *prefix* character as follows:

```
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:

```
10d      decimal
1010b    binary
12q      octal
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:   cmp         #’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 **substraction** operators only are accepted and only in the following cases:

```
label + constant
label - constant
label1 - label2
```

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
        clrx            ; prepare MSB
        tsta            ; test sign
        bpl     \@pos   ; if not positive
        comx            ; invert MSB
\@pos:  endm            ; end of macro
```

This macro is named *signex* and contains the code needed to perform a sign extension of **a** into **x**. Whenever needed, this macro can be expanded just by using its name in place of a standard instruction:

```
        lda     char+1 ; load LSB
        signex         ; expand macro
        stx     char   ; store result
```

The resulting code will be the same as if the following code had been written:

```
        lda     char+1 ; load LSB
        clrx           ; prepare MSB
        tsta           ; test sign
        bpl     pos    ; if not positive
        comx           ; invert MSB
pos:    stx     char   ; 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 catenate 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 the **numbered** syntax:

```
signex: macro         ; sign extension
        clrx          ; prepare MSB
        lda   \1      ; load LSB
        bpl   \@pos   ; if not positive
        comx          ; invert MSB
\@pos:  stx   \1      ; store MSB
        endm          ; end of macro
```

And called:

```
            signex char; sign extend char
```

This macro may also be written using the **named** syntax:

```
signex: macro \value    ; sign extension
        clrx            ; prepare MSB
        lda   \value    ; load LSB
        bpl   \@pos     ; if not positive
        comx            ; invert MSB
\@pos:  stx   \(value)  ; store MSB
        endm            ; end of macro
```

The form of a macro call is:

```
name>[.<ext>] [<arguments>]
```

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, all macros must then be defined before their first call. 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
ldx     #message
jsr     print
endif
```

If the symbol `debug` is equal to 1, the next two lines are assembled. Otherwise they are skipped.

```
if      offset != 1     ; if offset too large
addptr  offset          ; call a macro
else                    ; otherwise
incx                    ; increment X register
endif
```

If the symbol *offset* is not one, the macro `addptr` is expanded with *offset* as argument, otherwise the *aix* 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*.

```
data:  section          ; defines data section
text:  section          ; defines text section
start:
       ldx     #value   ; fills text section
```

```
        jmp     print
        switch  data     ; 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.

The assembler predefines the following sections, meaning that a *section* directive is *not* needed before to use them:

Section	Description
.text	executable code
.data	initialized data
.bss	uninitialized data
.bsct	initialized data in zero page
.ubsct	non initialized data in zero page

The sections **.bsct** and **.ubsct** are used for locating data in the zero page of the processor. The zero page is defined as the memory addresses between **0x00** and **0xFF** inclusive, *i.e.* the memory directly addressable by a single byte. Several processors include special instructions and/or addressing modes that take advantage of this special address range. The Cosmic assembler will automatically use the most efficient addressing mode if the data objects are allocated in the **.bsct**, **.ubsct** or a section with the same attributes. If zero page data objects are defined in another file then the directive **xref.b** must be used to externally reference the data object. This directive specifies that the address for these data object is only one byte and therefore the assembler may use 8 bit addressing modes.

```
xref        var
xref.b      zvar
      switch .bsct
zvar2:      ds.b 1
      switch .bss
var2:       ds.b 1
```

```
switch .text
lda   var
lda   zvar
lda   var2
lda   var2
end
```

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 **[-128,127].** 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
                               jmp   farlabel
```

Note that a *bra* instruction will be replaced by a single *jmp* instruction if it cannot be encoded as a relative branch.

A *jmp* or *jsr* instruction will be replaced by a *bra* or *bsr* instruction if the destination address is in the same section than the current one, and if the displacement is in the range allowed by a relative branch.

Old Syntax

The **-m** option allows the assembler to accept old constructs which are now obsolete. The following features are added to the standard behaviour:

- a comment line may begin with a '*****' character;
- a label starting in the first column does not need to be ended with a '**:**' character;
- no error message is issued if an operand of the **dc.b** directive is too large;
- the **section** directive handles *numbered* sections;

The comment separator at the end of an instruction is still the '**;**' character because the '*****' character is interpreted as the multiply operator.

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 *ca6805* 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
lda      #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

The *.bsct* section is limited to 256 bytes, but the assembler does not check the *.bsct* section size. This will be done by the linker.

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
endc
```

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               ; otherwise
incx               ; increment X register
endif
```

Note

The **else** and **elsec** directives are equivalent and may 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-127  ; if offset too large
addptr offset      ; call a macro
elsec              ; otherwise
incx               ; increment X register
endc
```

Note

The **elsec** and **else** directives are equivalent and may 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-127  ; if offset too large
addptr offset      ; call a macro
elsec              ; otherwise
incx               ; increment X register
endc
```

Note

The **endc** and **endif** directives are equivalent and may 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
incx               ; increment X register
endif
```

Note

The **endif** and **endc** directives are equivalent and may 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
        <repeat_body>
        endr
```

Function

The **endr** directive is used to terminate *repeat* sections.

Example

```
; shift a value n times
asln:   macro
        repeat \1
        aslb
        endr
        endm

; use of above macro
        asln 10     ;shift 10 times
```

See Also

repeat, repeatl

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

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:  equu   512
      ifge   value - Max
      fail   “Value too large”
```

if

Description

Conditional assembly

Syntax

```
if <expression>      or     if <expression>
instructions                instructions
endif                       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
incx                ; increment X register
endif
```

See Also

else, endif, clist

ifc

Description

Conditional assembly

Syntax

```
  ifc <string1>,<string2>  orifc <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
ldab     #45         ; load 45
elsec                ; otherwise...
ldab     #0
endc
```

See Also

elsec, endc, clist

ifdef

Description

Conditional assembly

Syntax

```
ifdef <label>      or      ifdef <label>
instructions               instructions
endc                       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>    or    ifeq <expression>
instructions               instructions
endc                       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
tsta                 ; just test it
elsec                ; otherwise
add     #offset      ; add to accu
endc
```

See Also

elsec, endc, clist

ifge

Description

Conditional assembly

Syntax

```
ifge <expression>    or   ifge <expression>
instructions              instructions
endc                      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-127  ; if offset too large
addptr offset      ; call a macro
elsec              ; otherwise
incx               ; increment X register
endc
```

See Also

elsec, endc, clist

ifgt

Description

Conditional assembly

Syntax

```
ifgt <expression>    or    ifgt <expression>
instructions               instructions
endc                       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-127   ; if offset too large
addptr offset       ; call a macro
elsec               ; otherwise
incx                ; increment X register
endc
```

See Also

elsec, endc, clist

ifle

Description

Conditional assembly

Syntax

```
ifle <expression>    or    ifle <expression>
instructions               instructions
endc                       elsec
                           instructions
                           endc
```

Function

The **ifle**, **elsec** and **endc** directives allow conditional assembly. The **ifle** 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 **ifle** 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 **ifle** 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

```
ifle    offset-127   ; if offset small enough
incx                 ; increment X register
elsec                ; otherwise
addptr  offset       ; call a macro
endc
```

See Also

elsec, endc, clist

iflt

Description

Conditional assembly

Syntax

```
iflt <expression>    or    iflt <expression>
instructions               instructions
endc                       elsec
                           instructions
                           endc
```

Function

The **iflt**, **else** 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-127  ; if offset small enough
incx               ; increment X register
elsec              ; otherwise
addptr offset      ; call a macro
endc
```

See Also

elsec, endc, clist

ifndef

Description

Conditional assembly

Syntax

```
ifndef <label>      or      ifndef <label>
instructions                instructions
endc                        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>    or    ifne <expression>
instructions               instructions
endc                       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     #offset       ; add to accu
elsec                 ; otherwise
tsta                  ; 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 differents, 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               ; otherwise
incx               ; increment X 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

local

Description

Create a new local block

Syntax

```
local
```

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$:        lda       var
            beq       10$
            sta       var2
local
10$:        lda       var2
            beq       10$
            sta       var
            rts
```

macro

Description

Define a macro

Syntax

```
  label: macro
         <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*"

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”
      lda   _#2
      sta   _SCR
```

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
       jsr \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 <expresion>
```

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 is 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
size:
      ldx    next       ; ends the offset section
```

org

Description

Sets the location counter to an offset from the beginning of a section.

Syntax

```
org <expresion>
```

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

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.

A **repeat** 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

```
; shift a value n times
asln: macro
      repeat \1
      aslb
      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    #\1        ; add to accu
      endr
      end
```

or

```
      ; test a value using the named syntax
      repeatl    \count,1,2,3
            add    #\count    ; add to accu
      endr
      end
```

will both produce:

```
 2           ; test a value
 9  0000 ab01   add    #1; add to accu
 9  0002 ab02   add    #2; add to accu
 9  0004 ab03   add    #3; add to accu
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$:  lda   var
      beq   10$
      sta   var2
function2:
10$:  lda   var2
      sub   var
      bne   10$
      rts
      restore
var3:       ds.b 1
var4:       ds.b 1


      switch .text

      lda   var3
      sta   var4
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
      aslb
      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$:    lda   var
        beq   10$
        sta   var2
function2:
10$:    lda   var2
        sub   var
        bne   10$
        rts
        restore
var3:         ds.b 1
var4:         ds.b 1


        switch .text

        lda   var3
        sta   var4
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 defined 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

Example

```
CODE:  section.text       ; section of text
lab1:  ds.b   5
DATA:  section.data       ; section of data
lab2:  ds.b   6
       switch CODE
lab3:  ds.b   7
       switch DATA
lab4:  ds.b   8
```

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:    section  1   ; define section 1
         nop
.ram:    section  2   ; define section 2
         dc.b     1
         section  1   ; 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

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

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

This chapter discusses the **clnk** linker and details how it operates. It describes each linker option, and explains how to use the linker's many special features. It also provides example linker command lines that show you how to perform some useful operations. This chapter includes the following sections:

Introduction

The linker combines relocatable object files, selectively loading from libraries of such files made with *clib*, to create an executable image for standalone execution or for input to other binary reformatters.

clnk will also allow the object image that it creates to have local symbol regions, so the same library can be loaded multiple times for different segments, and so that more control is provided over which symbols are exposed. On microcontroller architectures this feature is useful if your executable image must be loaded into several noncontiguous areas in memory.

NOTE

The terms "segment" and "section" refer to different entities and are carefully kept distinct throughout this chapter. A "section" is a contiguous subcomponent of an object module that the linker treats as indivisible.

The assembler creates several sections in each object module. The linker combines input sections in various ways, but will not break one up. The linker then maps these combined input sections into output segments in the executable image using the options you specify.

A "*segment*" is a logically unified block of memory in the executable image. An example is the code segment, which contains the executable instructions.

For most applications, the "*sections*" in an object module that the linker accepts as input are equivalent to the "segments" of the executable image that the linker generates as output.

Overview

You use the linker to build your executable program from a variety of modules. These modules can be the output of the C cross compiler, or can be generated from handwritten assembly language code. Some modules can be linked unconditionally, while others can be selected only as needed from function libraries. All input to the linker, regardless of its source, must be reduced to object modules, which are then combined to produce the program file.

The output of the linker can be in the same format as its input. Thus, a program can be built in several stages, possibly with special handling at some of the stages. It can be used to build freestanding programs such as system bootstraps and embedded applications. It can also be used to make object modules that are loaded one place in memory but are designed to execute somewhere else. For example, a data section in ROM to be copied into RAM at program startup can be linked to run at its actual target memory location. Pointers will be initialized and address references will be in place.

As a side effect of producing files that can be reprocessed, *clnk* retains information in the final program file that can be quite useful. The symbol table, or list of external identifiers, is handy when debugging programs, and the utility *cobj* can be made to produce a readable list of symbols from an object file. Finally, each object module has in its header useful information such as section sizes.

In most cases, the final program file created by *clnk* is structurally identical to the object module input to *clnk*. The only difference is that the executable file is complete and contains everything that it needs to run. There are a variety of utilities which will take the executable file and convert it to a form required for execution in specific microcontroller environments. The linker itself can perform some conversions, if all that is required is for certain portions of the executable file to be stripped off and for sections to be relocated in a particular way. You can therefore create executable programs using the linker that can be passed directly to a PROM programmer.

The linker works as follows:

- Options applying to the linker configuration. These options are referred to in this chapter as "*Global Command Line Options*" on page 233.
- Command file options apply only to specific sections of the object being built. These options are referred to in this chapter as "*Segment Control Options*" on page 234.
- Sections can be relocated to execute at arbitrary places in physical memory, or "stacked" on suitable storage boundaries one after the other.
- The final output of the linker is a header, followed by all the sections and the symbol table. There may also be an additional debug symbol table, which contains information used for debugging purposes.

Linker Command File Processing

The command file of the linker is a small control language designed to give the user a great deal of power in directing the actions of the linker. The basic structure of the command file is a series of command items. A command item is either an explicit linker option or the name of an input file (which serves as an implicit directive to link in that file or, if it is a library, scan it and link in any required modules of the library).

An explicit linker option consists of an option keyword followed by any parameters that the option may require. The options fall into five groups:

Group 1
(**+seg** *<section>*) controls the creation of new segments and has parameters which are selected from the set of local flags. (**+grp** *<section>*) controls the section grouping.
Group 2
(**+inc***) is used to include files
Group 3
(**+new**, **+pub** and **+pri**) controls name regions and takes no parameters.
Group 4
(**+def** *<symbol>*) is used to define symbols and aliases and takes one required parameter, a string of the form **ident1=ident2**, a string of the form **ident1=constant**, or a string of the form **ident1=@segment**.
Group 5
(**+spc** *<segment>*) is used to reserve space in a particular *<segment>* and has a required parameter

A description of each of these command line options appears below.

The manner in which the linker relocates the various sections is controlled by the **+seg** option and its parameters. If the size of a current segment is zero when a command to start a new segment of the same name is encountered, it is discarded. Several different sections can be redirected directly to the same segment by using the **+grp** option.

clnk links the <*files*> you specify in order. If a file is a library, it is scanned as long as there are modules to load. Only those library modules that define public symbols for which there are currently outstanding unsatisfied references are included.

Inserting comments in Linker commands

Each input line may be ended by a comment, which must be prefixed by a # character. If you have to use the # as a significant character, you can escape it, using the syntax \#.

Here is an example for an indirect link file:

```
# Link for EPROM
+seg .text -b0x1000 -n .text  # start eprom address
+seg .const -a .text          # constants follow program
+seg .bsct -b 0x20 -m 0x100   # zero page start address
+seg .ubsct -n iram           # uninitialized zero page
+seg .share -a iram -is       # shared segment
\cx32\lib\crts.h05            # startup object file
mod1.o mod2.o                 # input object files
\cx32\lib\libi.h05            # C library
\cx32\lib\libm.h05            # machine library
+seg .const -b0x3ff4          # vectors eprom address
vector.o                      # reset and interrupt vectors
```

Linker Options

The linker accepts the following options, each of which is described in detail below.

```
  clnk [options] <file.lkf> [<files>]
        -bs#   bank size
        -e*    error file name
        -l*>   library path
        -m*    map file name
        -o*    output file name
        -p     phys addr in map
        -s     symbol table only
        -v     verbose
```

The **output file name** and the **link command file** **must** be present on the command line. The options are described in terms of the two groups listed above; the global options that apply to the linker, and the segment control options that apply only to specific segments.

Global Command Line Options

The global command line options that the linker accepts are:

-bs#	set the window shift to #, which implies that the number of bytes in a window is **2**#**. The default value is 0 (bank switching disabled). For more information, see the section "*Address Arithmetic*" on page 242.
-e*	log errors in the text file ***** instead of displaying the messages on the terminal screen.
-l*>	specify library path. You can specify up to 20 different paths. Each path is a directory name, **not** terminated by any directory separator character.
-m*	produce map information for the program being built to file *.
-o*	write output to the file *. This option is required and has no default value.
-p	display symbols with physical address instead of logical address in the map file.
-s	create an output file containing only an absolute symbol table, but still with an object file format. The resulting file can then be used in another link to provide the symbol table of an existing application.
-v	be "verbose".

Segment Control Options

This section describes the segment control options that control the structure of individual segments of the output module.

A group of options to control a specific segment must begin with a **+seg** option. Such an option must precede any group of options so that the linker can determine which segment the options that follow apply to. The linker allows up to **255** different segments.

+seg <segment> <options> start a new segment with the name <*segment*> and build it as directed by the <*options*> that follow:

-a*	make the current segment follow the segment *. Options **-b** and **-o** cannot be specified if **-a** has been specified.
-b*	set the physical start address of the segment to *. Option **-e** cannot be specified if **-b** has been specified.
-c	do not output any code/data for the segment.
-ds#	set the bank size for paged addresses calculation. This option overwrites the global **-bs** option for that segment.
-e*	set the physical end address of the segment to *. Option **-b** cannot be specified if **-e** has been specified.
-f#	fill the segment up to the value specified by the **-m** option with bytes whose value is #. This option has no effect if no **-m** option is specified for that segment.
-i?	define the initialization option. Valid options are:

-it	use this segment to host the descriptor and images copies of initialized data used for automatic data initialization
-id	initialize this segment
-ib	do not initialize this segment
-is	mark this segment as shared data

-m*

set the maximum size of the segment to * bytes. If not specify, there is no checking on any segment size. If a segment is declared with the **-a** option as following a segment which is marked with the **-m** option, then set the maximum available space for all the possible consecutive segments.

-n*

set the output name of the segment to *. Segment output names have at most **15** characters; longer names are truncated. For example, use this option when you want to generate the hex records for a particular PROM, such as:

```
+seg .text -b0x2000 -n prom1
<object_files>
+seg .text -b0x4000 -n prom2
<object_files>
...
```

You can generate the hex records for **`prom1`** by typing:

```
chex -n prom1 file.h05
```

For more information, see "*The chex Utility*" in Chapter 8.

-o*

set the logical start address of the segment to * if **-b** option is specified or the logical end address if **-e** option is specified. The default is to set the logical address equal to the physical address. Options **-b** and **-e** cannot be specified both if **-o** has been specified.

-p*

define the bank number of the segment. This information will be used in case of bank switching instead of the computation based on the **-b** and **-bs** values.

-r*

round up the starting address of the segment. The expression defines the power of two of the alignment value. The option **-r3** will align the start address to an 8 bytes boundary. This option has no effect if the start address is explicitly defined by a **-b** option.

-s*	define a space name for the segment. This segment will be verified for overlapping only against segments defined with the same space name.
-v	do not verify overlapping for the segment.
-w*	set the window size for banked applications, and activate the automatic bank segment creation.
-x	expandable segment. Allow a segment to spill in the next segment of the same type if its size exceeds the value given by the **-m** option. The next segment must be declared **before** the object causing the overflow. This option has no effect if no **-m** option is specified for the expendable segment. Options **-e** and **-w** cannot be specified.

Options defining a numerical value (addresses and sizes) can be entered as constant, symbols, or simple expression combined them with '**+**' and '**-**' operators. Any symbol used has to be defined before to be used, either by a **+def** directive or loaded as an absolute symbol from a previously loaded object file. The operators are applied from left to right without any priority and parenthesis **()** are not allowed. Such expressions CANNOT contain any whitespace. For example:

```
+def START=0x1000
+def MAXSIZE=0x2000
+seg .text -bSTART+0x100 -mMAXSIZE-0x100
```

The first line defines the symbol `START` equals to the absolute value `1000` (hex value), the second line defines the symbol `MAXSIZE` equals to the absolute value `2000` (hex value). The last line opens a **.text** segment located at 1100 (hex value) with a maximum size of 1f00 (hex value). For more information, see the section "*Symbol Definition Option*" on page 240.

Unless **-b*** is given to set the *bss* segment start address, the *bss* segment will be made to follow the last *data* segment in the output file. Unless **-b*** is given to set the *data* segment start address, the *data* segment will be made to follow the last *bsct* segment in the output file. The *bsct* and *text* sections are set to start at zero unless you specify otherwise by using **-b** option. It is permissible for all segments to overlap, as far as *clnk* is concerned; the target machine may or may not make sense of this situation (as with separate instruction and data spaces).

NOTE

A new segment of the specified type will not actually be created if the last segment of the same name has a size of zero. However, the new options will be processed and will override the previous values.

Segment Grouping

Different sections can be redirected directly to the same segment with the **+grp** directive:

+grp <section>=<section list>

where <*section*> is the name of the target section, and <*section list*> a list of section names separated by commas. When loading an object file, each section listed in the right part of the declaration will be loaded as if it was named as defined in the left part of the declaration. The target section may be a new section name or the name of an existing section (including the predefined ones). When using a new name, this directive has to be preceded by a matching **+seg** definition.

NOTE

*Whitespaces are **not** allowed aside the equal sign '=' and the commas.*

Linking Files on the Command line

The linker supports linking objects from the command line. The link command file has to be modified to indicate where the objects are to be loaded using the following **@#** syntax.

@1, @2,...	include each individual object file at its positional location on the command line and insert them at the respective locations in the link file (**@1** is the first object file, and so on).
@*	include all of the objects on the command line and insert them at this location in the link file.

Example

Linking objects from the command line:

```
clnk -o test.h05 test.lkf file1.o file2.o

## Test.lkf:
+seg  .text -b0x5000
+seg .data -b0x100
@1
+seg .text -b0x7000
@2

Is equivalent to

clnk -otest.h05 test.lkf
## test.lkf
+seg  .text -b0x5000
+seg .data -b0x100
file1.o
+seg .text -b0x7000
file2.o
```

Include Option

Subparts of the link command file can be included from other files by using the following option:

+inc*	include the file specified by *. This is equivalent to expanding the text file into the link file directly at the location of the **+inc** line.

Example

Include the file “`seg2.txt`” in the link file “`test.lkf`”:

```
## Test.lkf:
+seg  .text -b0x5000
+seg .data -b0x100
file1.o file2.o
+seg .text -b0x7000
+inc seg2.txt

## seg2.txt:
mod1.o mod2.o mod3.o

## Resultant link file
+seg  .text -b0x5000
+seg .data -b0x100
file1.o  file2.o
+seg .text -b0x7000
mod1.o mod2.o mod3.o
```

Private Region Options

Options that control code regions are:

+new	start a new region. A “region” is a user definable group of input object modules which may have both public and private portions. The private portions of a region are local to that region and may not access or be accessed by anything outside the region. By default, a new region is given public access.
+pub	make the following portion of a given region public.
+pri	make the following portion of a given region private.

Symbol Definition Option

The option controlling symbol definition and aliases is:

+def* define new symbols to the linker. The string ***** must be of the form:

- **ident1=ident2** where *ident1* and *ident2* are both valid identifiers. This form is used to define aliases. The symbol *ident1* is defined as the alias for the symbol *ident2* and goes in the symbol table as an external DEF (a DEF is an entity defined by a given module.) If *ident2* is not already in the symbol table, it is placed there as a REF (a REF is an entity referred to by a given module).

- **ident=@section** where *ident* is a valid identifier, and *section* is the name of a section specified as the first argument of a **+seg** directive. This form is used to add *ident* to the symbol table as a defined symbol whose value is the address of the next byte to be loaded in the specified section.

- **ident=constant** where *ident* is a valid identifier and *constant* is a valid constant expressed with the standard C language syntax. This form is used to add *ident* to the symbol table as a defined absolute symbol with a value equal to *constant*.

- **ident=start(segment)** where *segment* is the name given to a segment by the **-n** option. This form is used to add *ident* to the symbol table as a defined symbol whose value is the *logical* start address of the designated segment. This directive can be placed anywhere in the link command file, even before the segment is defined.

- **ident=end(segment)** where *segment* is the name given to a segment by the **-n** option. This form is used to add *ident* to the symbol table as a defined symbol whose value is the *logical* end address of the designated segment. This directive can be placed anywhere in the link command file, even before the segment is defined.

- **ident=pstart(segment)** where *segment* is the name given to a segment by the **-n** option. This form is used to add *ident* to the symbol table as a defined symbol whose value is the *physical* start address of the designated segment. This directive can be placed anywhere in the link command file, even before the segment is defined.

- **ident=pend(segment)** where *segment* is the name given to a segment by the **-n** option. This form is used to add *ident* to the symbol table as a defined symbol whose value is the *physical* end address of the designated segment. This directive can be placed anywhere in the link command file, even before the segment is defined.

- **ident=size(segment)** where *segment* is the name given to a segment by the **-n** option. This form is used to add *ident* to the symbol table as a defined symbol whose value is the size of the designated segment. This directive can be placed anywhere in the link command file, even before the segment is defined.

NOTE

*Whitespaces are **not** allowed aside the equal sign '='.*

For more information about DEFs and REFs, refer to the section "*DEFs and REFs*" on page 248.

Reserve Space Option

The following option is used to reserve space in a given segment:

+spc <segment>=<value> reserve <*value*> bytes of space at the current location in the segment named <*segment*>.

+spc <segment>=@section reserve a space at the current location in the segment named <*segment*> equal to the current size of the opened segment where the given *section* is loaded. The size is evaluated at once, so if the reference segment grows after that directive, there is no further modification of the space reservation. If such a directive is used to duplicate an existing section, it has to be placed in the link command file after all the object files.

NOTE

*Whitespaces are **not** allowed aside the equal sign '='.*

Section Relocation

The linker relocates the sections of the input files into the segments of the output file.

An absolute section, by definition, cannot and should not be relocated. The linker will detect any conflicts between the placement of this file and its absolute address given at compile/assemble time.

In the case of a bank switched system, it is still possible for an absolute section to specify a physical address different from the one and at compile/assembly time, the logical address MUST match the one specified at compile/assemble time.

Address Arithmetic

The two most important parameters describing a segment are its **bias** and its **offset**, respectively its physical and logical start addresses. In nonsegmented architectures there is no distinction between *bias* and *offset*. The *bias* is the address of the location in memory where the section is relocated to run. The *offset* of a segment will be equal to the *bias*. In this case you must set only the *bias*. The linker sets the *offset* automatically.

In segmented architectures, the fundamental relationship between the bias and the offset is:

```
bias = (SR << BS) + offset
```

where **SR** is the actual value used in a segment or page register and **BS** is the window shift value you specify with the **-bs#** option. The linker will be able to compute the value of the page register, given the bias and the offset of any section.

In nonsegmented architectures both **BS** and **SR** are usually equal to zero, so the formula becomes:

```
bias = offset
```

Overlapping Control

The linker is verifying that a segment does not overlap any other one, by checking the physical addresses (*bias*). This control can be locally disabled for one segment by using the **-v** option. For targets implementing separated address spaces (such as bank switching), the linker allows several segments to be isolated from the other ones, by giving them a *space* name with the **-s** option. In such a case, a segment in a named space is checked only against the other segments of the same space. The unnamed segments are checked together.

Setting Bias and Offset

The bias and offset of a section are controlled by the **-b*** option and **-o*** option. The rules for dealing with these options are described below.

Setting the Bias

If the **-b*** option is specified, the bias is set to ##. Otherwise, the bias is set to the end of the last segment of the same name.

Setting the Offset

If the **-o*** option is specified, the offset is set to ##. Otherwise, the offset is set equal to the bias.

Using Default Placement

If none of the **-b** and **-o** options are specified, the segment may be placed *after* another one, by using the **-a*** option, where ***** is the name of another segment. Otherwise, the linker will try to use a default placement based on the segment name. The compiler produces specific sections for code (*.text*) and data (*.data, .bss*, *.bsct* and *.ubsct*). By default, *.text* and *.bsct* segments start at zero, *.ubsct* segment follows the latest *.bsct* segment, *.data* segment follows the latest *.ubsct* segment, and *.bss* segment follows the latest *.data* segment. Note that there is no default placement for the constants sections *.const*.

Linking Objects

A new segment is built by concatenating the corresponding sections of the input object modules in the order the linker encounters them. As each input section is added to the output segment, it is adjusted to be relocated relative to the end portion of the output segment so far constructed. The first input object module encountered is relocated relative to a value that can be specified to the linker. The size of the output *bss* section is the sum of the sizes of the input *bss* sections.

Unless the **-v** option has been specified on a segment definition, the linker checks that the segment physical address range does not overlap any other segment of the application. Logical addresses are not checked as bank switching creates several segments starting at the same logical 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 place ***after*** all objects that may call it's 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.h05 Single Precision 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.h05 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.

Library Order

You should link your application with the libraries in the following orders:

Integer Only Application	Single Precision Float Application
libi.h05	libf.h05
libm.h05	libi.h05
	libm.h05

For more information, see “*Linker Command Line Examples*” on page 255.

Automatic Data Initialization

The linker is able to configure the executable for an automatic data initialization. This mechanism is initiated automatically when the linker finds the symbol **__idesc__** in the symbol table, as an *undefined* symbol. *clnk* first locates a segment behind which it will add an image of the data, so called the *host* segment. The default behaviour is to select the first *.text* segment in the executable file, but you can override this by marking one segment with the **-it** option.

Then, *clnk* looks in the executable file for initialized segments. All the segments *.data* and *.bsct* are selected by default, unless disabled explicitly by the **-ib** option. Otherwise, renamed segments may also be selected by using the **-id** option. The **-id** option cannot be specified on a bss segment, default or renamed. Once all the selected segments are located, *clnk* builds a descriptor containing the starting address and length of each such segment, and moves the descriptor and the selected segments to the end of the *host* segment, without relocating the content of the selected segments.

For more information, see "*Generating Automatic Data Initialization*" in Chapter 2, "*Tutorial Introduction*" and "*Initializing data in RAM*" in Chapter 3, "*Programming Environments*".

Descriptor Format

The created descriptor has the following format:

```
 dc.w start_prom_address  ;starting address of the
                          ; first image in prom
; for each segment:
 dc.b flag                ; segment type
 dc.w start_ram_address   ; start address of segment in ram
 dc.w end_prom_address    ; address of last data byte
                          ; plus one in prom
; after the last segment:
 dc.b 0
```

The flag byte is used to detect the end of the descriptor, and also to specify a type for the data segment. The actual value is equal to the code of the first letter in the segment name.

The end address in PROM of one segment gives also the starting address in prom of the following segment, if any.

The address of the descriptor will be assigned to the symbol *__idesc__*, which is used by the *crtsi.s* startup routine. So all this mechanism will be activated just by linking the *crtsi.h05* file with the application, or by referencing the symbol *__idesc__* in your own startup file.

If the *host* segment has been opened with a **-m** option giving a maximum size, *clnk* will check that there is enough space to move all the selected segments.

Shared Data Handling

When the compiler is run with a *static model* allowing *shared data*, each function not using the stack (*nostack* function) reserves a memory area but does not allocate it. Based on the information received from the debug symbol table, the linker is able to allocate all these areas by overlapping those areas corresponding to independent functions, *i.e.* these functions that never call each other directly or through other functions. This feature saves memory while keeping the ability to use arguments and local variables in *nostack* functions.

NOTE

Recursive functions cannot be selected as nostack functions.

The linker will allocate the global amount of memory in a segment provided by the user. This segment is marked with the **-is** option, and has to be empty. It is located like all other segments using either **-b** or **-a** options, as shown in the following example:

```
  +seg .text -b0x1000
  +seg .data -b0x100
  +seg .shared -b0x80 -m0x80 -is
  file1 file2
```

Object files *file1* and *file2* should not produce any data in a *.shared* section, otherwise the linker will complain and abort the linking process.

DEFs and REFs

The linker builds a new symbol table based on the symbol tables in the input object modules, but it is not a simple concatenation with adjustments. There are two basic type of symbols that the linker puts into its internal symbol table: **REF**s and **DEF**s. DEFs are symbols that are defined in the object module in which they occur. REFs are symbols that are referenced by the object module in which they occur, but are not defined there.

The linker also builds a debug symbol table based on the debug symbol tables in any of the input object modules. It builds the debug symbol table by concatenating the debug symbol tables of each input object module in the order it encounters them. If debugging is not enabled for any of input object module, the debug symbol table will be of zero length.

An incoming REF is added to the symbol table as a REF if that symbol is not already entered in the symbol table; otherwise, it is ignored (that reference has already been satisfied by a DEF or the reference has already been noted). An incoming DEF is added to the symbol table as a DEF if that symbol is not already entered in the symbol table; its value is adjusted to reflect how the linker is relocating the input object module in which it occurred. If it is present as a REF, the entry is changed to a DEF and the symbol's adjusted value is entered in the symbol table entry. If it is present as a DEF, an error occurs (multiply defined symbol).

When the linker is processing a library, an object module in the library becomes an input object module to the linker only if it has at least one DEF which satisfies some outstanding REF in the linker's internal symbol table. Thus, the simplest use of *clnk* is to combine two files and check that no unused references remain.

The executable file created by the linker must have no REFs in its symbol table. Otherwise, the linker emits the error message "*undefined symbol*" and returns failure.

Special Topics

This section explains some special linker capabilities that may have limited applicability for building most kinds of microcontroller applications.

Private Name Regions

Private name regions are used when you wish to link together a group of files and expose only some to the symbol names that they define. This lets you link a larger program in groups without worrying about names intended only for local usage in one group colliding with identical names intended to be local to another group. Private name regions let you keep names truly local, so the problem of name space pollution is much more manageable.

An explicit use for private name regions in an MC68HC05 environment is in building a paged program with duplication of the most used library functions in each page, in order to avoid extra page commutation. To avoid complaints when multiple copies of the same file redefine symbols, each such contribution is placed in a private name region accessible only to other files in the same page.

The basic sequence of commands for each island looks like:

```
+new <public files> +pri <private libraries>
```

Any symbols defined in <*public files*> are known outside this private name region. Any symbols defined in <*private libraries*> are known only within this region; hence they may safely be redefined as private to other regions as well.

NOTE

All symbols defined in a private region are local symbols and will not appear in the symbol table of the output file.

Renaming Symbols

At times it may be desirable to provide a symbol with an alias and to hide the original name (*i.e.*, to prevent its definition from being used by the linker as a DEF which satisfies REFs to that symbol name). As an example, suppose that the function *func* in the C library provided with the compiler does not do everything that is desired of it for some special application. There are three methods of handling this situation (we will ignore the alternative of trying to live with the existing function's deficiencies).

The first method is to write a new version of the function that performs as required and link it into the program being built before linking in the libraries. This will cause the new definition of *func* to satisfy any references to that function, so the linker does not include the version from the library because it is not needed. This method has two major drawbacks: first, a new function must be written and debugged to provide something which basically already exists; second, the details of exactly what the function must do and how it must do it may not be available, thus preventing a proper implementation of the function.

The second approach is to write a new function, say *my_func*, which does the extra processing required and then calls the standard function *func*. This approach will generally work, unless the original function *func* is called by other functions in the libraries. In that case, the extra function behavior cannot occur when *func* is called from library functions, since it is actually *my_func* that performs it.

The third approach is to use the aliasing capabilities of the linker. Like the second method, a new function will be written which performs the new behavior and then calls the old function. The twist is to give the old function a new name and hide its old name. Then the new function is given the old function's name and, when it calls the old function, it uses the new name, or alias, for that function. The following linker script provides a specific example of this technique for the function *func*:

```
line 1 +seg .text -b 0x1000
line 2 +seg .data -b0
line 3 +new
line 4 Crts.xx
line 5 +def _oldfunc=_func
line 6 +pri func.o
line 7 +new
line 8 prog.o newfunc.o
line 9 <libraries>
```

NOTE

The function name func as referenced here is the name as seen by the C programmer. The name which is used in the linker for purposes of aliasing is the name as seen at the object module level. For more information on this transformation, see the section "Interfacing C to Assembly Language" in Chapter 3.

The main thing to note here is that *func.o* and *new_func.o* both define a (different) function named *func*. The second function *func* defined in *newfunc.o* calls the old *func* function by its alias *oldfunc*.

Name regions provide limited scope control for symbol names. The **+new** command starts a new name region, which will be in effect until the next **+new** command. Within a region there are public and private name spaces. These are entered by the **+pub** and **+pri** commands; by default, **+new** starts in the public name space.

Lines 1,2 are the basic linker commands for setting up a separate I/D program. Note that there may be other options required here, either by the system itself or by the user.

Line 3 starts a new region, initially in the public name space.

Line 4 specifies the startup code for the system being used.

Line 5 establishes the symbol *_oldfunc* as an alias for the symbol *_func*. The symbol *_oldfunc* is entered in the symbol table as a public definition. The symbol *_func* is entered as a private reference in the current region.

Line 6 switches to the private name space in the current region. Then *func.o* is linked and provides a definition (private, of course) which satisfies the reference to *_func*.

Line 7 starts a new name region, which is in the public name space by default. Now no reference to the symbol *_func* can reach the definition created on **Line 6**. That definition can only be reached now by using the symbol *_oldfunc*, which is publicly defined as an alias for it.

Line 8 links the user program and the module *newfunc.o*, which provides a new (and public) definition of *_func*. In this module the old version is accessed by its alias. This new version will satisfy all references to *_func* made in *prog.o* and the libraries.

Line 9 links in the required libraries.

The rules governing which name space a symbol belongs to are as follows:

- Any symbol definition in the public space is public and satisfies all outstanding and future references to that symbol.
- Any symbol definition in the private space of the current region is private and will satisfy any private reference in the current region.
- All private definitions of a symbol must occur before a public definition of that symbol. After a public definition of a symbol, any other definition of that symbol will cause a "*multiply defined symbol*" error.
- Any number of private definitions are allowed, but each must be in a separate region to prevent a multiply defined symbol error.
- Any new reference is associated with the region in which the reference is made. It can be satisfied by a private definition in that region, or by a public definition. A previous definition of that symbol will satisfy the reference if that definition is public, or if the definition is private and the reference is made in the same region as the definition.
- If a new reference to a symbol occurs, and that symbol still has an outstanding unsatisfied reference made in another region, then that symbol is marked as requiring a public definition to satisfy it.
- Any definition of a symbol must satisfy all outstanding references to that symbol; therefore, a private definition of a symbol which requires a public definition causes a blocked symbol reference error.

- No symbol reference can “reach” any definition made earlier than the most recent definition.

Absolute Symbol Tables

Absolute Symbol tables are used to export symbols from one application to another, to share common functions for instance, or to use functions already built in a ROM, from an application downloaded into RAM. The linker option **-s** will modify the output file in order to contain only a symbol table, without any code, but still with an object file format, by using the same command file used to build the application itself. All symbols are flagged as *absolute* symbols. This file can be used in another link, and will then transmit its symbol table, allowing another application to use those symbols as *externals*. Note that the linker does not produce any map even if requested, when used with the **-s** option.

The basic sequence of commands looks like:

```
clnk -o appli.h05 -m appli.map appli.lkf
clnk -o appli.sym  -s appli.lkf
```

The first link builds the application itself using the *appli.lkf* command file. The second link uses the same command file and creates an object file containing only an absolute symbol table. This file can then be used as an input object file in any other link command file.

Description of The Map File

The linker can output a map file by using the **-m** option. The map file contains 4 sections: the *Segment* section, the *Modules* section, the *Stack Usage* section and the *Symbols* section.

Segment Describe the different segments which compose the application, specifying for each of them: the start address (in hexa), the end address (in hexa), the length (in decimal), and the name of the segment. Note that the end value is the address of the byte following the last one of the segment, meaning that an empty segment will have the same start and end addresses. If a segment is initialized, it is displayed twice, the first time with its final address, the second time with the address of the image copy.

Modules List all the modules which compose the application, giving for each the description of all the defined sections with the same format as in the *Segment* section. If an object has been assembled with the **-pl** option, local symbols are displayed just after the module description.

Stack Usage Describe the amount of memory needed for the stack. When using a **stack model**, each function of the application is listed by its name, followed by a '>' character indicating that this function is not called by any other one (the *main* function, *interrupt* functions, *task* entries...). The first number is the total size of the stack used by the function including all the internal calls. The second number between braces shows the stack need for that function alone. When using a **memory model** each function using space in the simulated stack is listed by its name followed by the address range of its local area, and followed by two numbers between braces. The first one indicates how many bytes are used for locals and the second one indicates how many bytes are used for arguments. Functions locally redirected to the physical stack are also displayed with their stack usage. The linker displays at the end of the list a total stack size assuming interrupt functions cannot be themselves interrupted. Interrupt frames and machine library calls are properly counted.

Symbols List all the symbols defined in the application specifying for each its name, its value, the section where it is defined, and the modules where it is used. If the target processor supports bank switching, addresses are displayed as logical addresses by default. Physical addresses can be displayed by specifying the **-p** option on the linker command line.

Return Value

clnk returns success if no error messages are printed to STDOUT; that is, if no undefined symbols remain and if all reads and writes succeed. Otherwise it returns failure.

Linker Command Line Examples

This section shows you how to use the linker to perform some basic operations.

A linker command file consists of linker options, input and output file, and libraries. The options and files are read from a command file by the linker. For example, to create an MC68HC05 file from *file.o* you can type at the system prompt:

```
  clnk -o myapp.h05 myapp.lkf
```

where *myapp.lkf* contains:

```
+seg .text -b0x1000 -n .text  # start eprom address
+seg .const -a .text          # constants follow program
+seg .bsct -b0x20 -n iram -m 0x100# initialized zero page
+seg .share -a iram -is       # shared segment
\cx32\lib\crts.h05            # startup object file
file1.o file2.o               # input object files
\cx32\lib\libi.h05            # C library
\cx32\lib\libm.h05            # machine library
+def __memory=@.bss           # symbol used by startup
```

The following link command file is an example for an application that does **not** use floating point data types and does **not** require automatic initialization.

```
# demo.lkf: link command WITHOUT automatic init
+seg .text -b 0x1000 -n.text  # program start address
+seg .const -a .text          # constants follow program
+seg .bsct -b0x20 -n iram -m0x100# zpage start address
+seg .share -a iram -is       # shared segment
+seg .data -b0x100            # start data address
\cx32\lib\crts.h05            # startup with NO-INIT
acia.o                        # main program
module1.o                     # module program
\cx32\lib\libi.h05            # C lib.
\cx32\lib\libm.h05            # machine lib.
+seg .const -b0xff4           # vectors eprom address
vector.o                      # reset and interrupt vectors
+def __memory=@.bss           # symbol used by library
```

The following link command file is an example for an application that uses single precision floating point data types and utilizes automatic data initialization.

```
# demo.lkf: link command WITH automatic init
+seg .text -b 0x1000 -n.text  # program start address
+seg .const -a .text          # constants follow program
+seg .bsct -b0x20 -n iram -m0x100# zpage start address
+seg .share -a iram -is       # shared segment
+seg .data -b0x100            # start data address
\cx32\lib\crtsi.h05           # startup with auto-init
acia.o                        # main program
module1.o                     # module program
\cx32\lib\libf.h05            # single prec.
\cx32lib\libi.h05             # integer lib.
\cx32\lib\libm.h05            # machine lib.
+seg .const -b0x3ff4          # vectors eprom address
vector.o                      # reset and interrupt vectors
+def __memory=@.bss           # end of bss segment
```

CHAPTER
7

Debugging Support

This chapter describes the debugging support available with the cross compiler targeting the MC68HC05. There are two levels of debugging support available, so you can use either the COSMIC's **Zap** C source level cross debugger or your own debugger or in-circuit emulator to debug your application. This chapter includes the following sections:

Generating Debugging Information

The compiler generates debugging information in response to command line options you pass to the compiler as described below. The compiler can generate the following debugging information:

1 line number information that allows COSMIC's C source level debugger or another debugger or emulator to locate the address of the code that a particular C source line (or set of lines) generates. You may put line number information into the object module in either of the two formats, or you can generate both line number information and information about program data and function arguments, as described below.

2 information about the name, type, storage class and address (absolute or relative to a stack offset) of program static data objects, function arguments, and automatic data objects that functions declare. Information about what source files produced which relocatable or executable files. This information may be localized by address (where the output file resides in memory). It may be written to a file, sorted by address or alphabetical order, or it may be output to a printer in paginated or unpaginated format.

Generating Line Number Information

The compiler puts line number information into a special debug symbol table. The debug symbol table is part of the relocatable object file produced by a compilation. It is also part of the output of the *clnk* linker. You can therefore obtain line number information about a single file, or about all the files making up an executable program. However, the compiler can produce line number information only for files that are **fewer** than 65,535 lines in length.

Generating Data Object Information

The **+debug** option directs the compiler to generate information about data objects and function arguments and return types. The debugging information the compiler generates is the information used by the COSMIC's C source level cross debugger or another debugger or emulator. The information produced about data objects includes their name, scope, type and address. The address can be either absolute or relative to a stack offset.

As with line number information alone, you can generate debugging information about a single file or about all the files making up an executable program.

cprd may be used to extract the debugging information from files compiled with the **+debug** option, as described below.

The cprd Utility

cprd extracts information about functions and data objects from an object module or executable image that has been compiled with the **+debug** option. *cprd* extracts and prints information on the name, type, storage class and address (absolute or offset) of program static data objects, function arguments, and automatic data objects that functions declare. For automatic data, the address provided is an offset from the frame pointer. For function arguments, the address provided is an offset from the stack pointer.

Command Line Options

cprd accepts the following command line options, each of which is described in detail below:

```
cprd [options] file
     -fc*   select function name
     -fl*   select file name
     -o*    output file name
```

where <*file*> is an object file compiled from C source with the compiler command line option **+debug** set.

-fc* print debugging information only about the function *. By default, *cprd* prints debugging information on all functions in <*file*>. Note that information about global data objects is always displayed when available.

-fl* print debugging information only about the file *. By default, *cprd* prints debugging information on all C source files.

-o* print debugging information to file *. Debugging information is written to your terminal screen by default.

By default, *cprd* prints debugging information about all functions and global data objects in <*file*>.

Examples

The following example show sample output generated by running the *cprd* utility on an object file created by compiling the program *acia.c* with the compiler option **+debug** set.

```
   cprd acia.h05
```

```
Information extracted from acia.h05

source file acia.c:

(no globals)

unsigned char getch() lines 25 to 35 at 0xf016-0xf030
    auto unsigned char c at -1 from frame pointer

void outch() lines 39 to 44 at 0xf031-0xf03d
    argument unsigned char c at 3 from frame pointer

void recept() lines 50 to 56 at 0xf03e-0xf113
    (no locals)

void main() lines 62 to 71 at 0xf114-0xf06b
    (no locals)
```

The clst utility

The **clst** utility takes relocatable or executable files as arguments, and creates listings showing the C source files that were compiled or linked to obtain those relocatable or executable files. It is a convenient utility for finding where the source statements are implemented.

To use *clst* efficiently, its argument files must have been compiled with the **+debug** option.

clst can be instructed to limit its display to files occupying memory in a particular range of addresses, facilitating debugging by excluding extraneous data. *clst* will display the entire content of any files located between the endpoints of its specified address range.

Command Line Options

clst accepts the following command line options, each of which is described in detail below:

```
 clst [options> file
      -a   list file alphabetically
      -f*> process selected file
      -i*> source file
      -l#  page length
      -o*  output file name
      -p   suppress pagination
      -r*  specify a line range #:#
```

-a when set, cause *clst* to list files in alphabetical order. The default is that they are listed by increasing addresses.

-f*> specify * as the file to be processed. Default is to process all the files of the application. Up to 10 files can be specified.

-i*> read string * to locate the source file in a specific directory. Source files will first be searched for in the current directory, then in the specified directories in the order they were given to *clst*. You can specify up to 20 different paths Each path is a directory name, **not** terminated by any directory separator character.

-l# when paginating output, make the listings # lines long. By default, listings are paginated at 66 lines per page.

-o* redirect output from *clst* to file *****. You can achieve a similar effect by redirecting output in the command line.

```
clst -o acia.lst acia.h05
```

is equivalent to:

```
clst acia.h05 >acia.lst
```

-p suppress pagination. No page breaks will be output.

-r#:# where **#:#** is a range specification. It must be of the form *<number>:<number>*. When this flag is specified, only those source files occupying memory in the specified range will be listed. If part of a file occupies memory in the specified range, that file will be listed in its entirety. The following is a valid use of **-r**:

```
-r 0xe000:0xe200
```

CHAPTER
8

Programming Support

This chapter describes each of the programming support utilities packaged with the C cross compiler targeting the MC68HC05. The following utilities are available:

chex	translate object module format
clabs	generate absolute listings
clib	build and maintains libraries
cobj	examine objects modules
cv695	generate IEEE695 format
cvdwarf	generate ELF/DWARF format

The assembler is described in Chapter 5, “*Using The Assembler*”. The linker is described in Chapter 6, “*Using The Linker*”. Support for debugging is described in Chapter 7, “*Debugging Support*”.

The description of each utility tells you what tasks it can perform, the command line options it accepts, and how you use it to perform some commonly required operations. At the end of the chapter are a series of examples that show you how to combine the programming support utilities to perform more complex operations.

The chex Utility

You use the **chex** utility to translate executable images produced by *clnk* to one of several hexadecimal interchange formats. These formats are: *Motorola S-record* format, and *Intel standard hex* format. You can also use *chex* to override text and data biases in an executable image or to output only a portion of the executable.

The executable image is read from the input file <*file*>.

Command Line Options

chex accepts the following command line options, each of which is described in detail below:

```
  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
       -pn    use paged address in bank only
       -pp    use paged address with mapping
       -s     output increasing addresses
       -x*>   exclude named segments
```

-a##	the argument file is a considered as a pure binary file and ## is the output address of the first byte.
-b##	substract ## to any address before output.
-e##	define ## as the entry point address encoded in the dedicated record of the output format, if available.
-f?	define output file format. Valid options are:

i	Intel hex format
m	Motorola S19 format
2	Motorola S2 format
3	Motorola S3 format

Default is to produced Motorola S-Records (**-fm**). Any other letter will select the default format.

-h	do not output the header sequence if such a sequence exists for the selected format.
+h*	insert ***** in the header sequence if such a sequence exists for the selected format.
-m#	output **#** maximum data bytes per line. Default is to output 32 bytes per line.
-n*>	output only segments whose name is equal to the string *****. Up to twenty different names may be specified on the command line. If there are several segments with the same name, they will all be produced. This option is used in combination with the **-n** option of the linker.
-o*	write output module to file *****. The default is STDOUT.
-p	output addresses of banked segments using a paged format `<page_number><logical_address>`, instead of the default format `<physical>`.
-pn	behaves as **-p** but only when logical address is inside the banked area. This option has to be selected when producing an hex file for the Noral debugger.
-pp	behaves as **-p** but uses paged addresses for all banked segments, mapped or unmapped. This option has to be selectd when producing an hex file for Promic tools.
-s	sort the output addresses in increasing order.

-x*> do not output segments whose name is equal to the string *. Up to twenty different names may be specified on the command line. If there are several segments with the same name, they will not all be output.

Return Status

chex returns success if no error messages are printed; that is, if all records are valid and all reads and writes succeed. Otherwise it returns failure.

Examples

The file *hello.c*, consisting of:

```
char *p = {“hello world”};
```

when compiled produces the following the following *Motorola S-record* format:

```
chex hello.o
```

```
S00A000068656C6C6F2E6F44
S1110000020068656C6C6F20776F726C640090
S9030000FC
```

and the following *Intel standard hex* format:

```
chex -fi hello.o
```

```
:0E000000020068656C6C6F20776F726C640094
:00000001FF
```

The clabs Utility

clabs processes assembler listing files with the associated executable file to produce listing with updated code and address values.

clabs decodes an executable file to retrieve the list of all the files which have been used to create the executable. For each of these files, *clabs* looks for a matching listing file produced by the compiler ("**.ls**" file). If such a file exists, *clabs* creates a new listing file ("**.la**" file) with absolute addresses and code, extracted from the executable file.

To be able to produce any results, the compiler **must** have been used with the '**-l**' option.

Command Line Options

clabs accepts the following command line options, each of which is described in detail below.

```
  clabs [options] file
        -a     process also library files
        -l     restrict to local directory
        -p     use paged address format
        -pn    use paged address in bank only
        -pp    use paged address with mapping
        -r*    relocatable listing suffix
        -s*    absolute listing suffix
        -v     echo processed file names
```

-a	process also files located in libraries. Default is to process only all the files of the application.
-l	process files in the current directory only. Default is to process all the files of the application.
-p	output addresses of banked segments using a paged format `<page_number><logical_address>`, instead of the default format `<physical>`.
-pn	behaves as **-p** but only when logical address is inside the banked area.

-pp	behaves as **-p** but uses paged addresses for all banked segments, mapped or unmapped.
-r*	specify the input suffix, including or not the dot '.' character. Default is "**.ls**"
-s*	specify the output suffix, including or not the dot '.' character. Default is "**.la**"
-v	be verbose. The name of each module of the application is output to STDOUT.

<*file*> specifies one file, which must be in executable format.

Return Status

clabs returns success if no error messages are printed; that is, if all reads and writes succeed. An error message is output if no relocatable listing files are found. Otherwise it returns failure.

Examples

The following command line:

```
clabs -v acia.h05
```

will output:

```
crts.ls
acia.ls
vector.ls
```

and creates the following files:

```
crts.la
acia.la
vector.la
```

The following command line:

```
clabs -r.lst acia.h05
```

will look for files with the suffix "**.lst**":

The following command line:

```
clabs -s.lx acia.h05
```

will generate:

```
crts.lx
acia.lx
vector.lx
```

The clib Utility

clib builds and maintains object module libraries. *clib* can also be used to collect arbitrary files in one place. <*library*> is the name of an existing library file or, in the case of replace or create operations, the name of the library to be constructed.

Command Line Options

clib accepts the following command line options, each of which is described in detail below:

```
  clib [options] <library> <files>
        -c     create a new library
        -d     delete modules from library
        -i*    object list filename
        -l     load all library at link
        -r     replace modules in library
        -s     list symbols in library
        -t     list files in library
        -v     be verbose
        -x     extract modules from library
```

-c create a library containing <*files*>. Any existing <*library*> of the same name is removed before the new one is created.

-d delete from the library the zero or more files in <*files*>.

-i* take object files from a list *****. You can put several files per line or put one file per line. Each lines can include comments. They must be prefixed by the '#' character. If the command line contains <*files*>, then <*files*> will be also added to the library.

-l when a library is built with this flag set, all the modules of the library will be loaded at link time. By default, the linker only loads modules necessary for the application.

-r in an existing library, replace the zero or more files in <*files*>. If no library <*library*> exists, create a library containing <*files*>. The files in <*files*> not present in the library are added to it.

-s	list the symbols defined in the library with the module name to which they belong.
-t	list the files in the library.
-v	be verbose
-x	extract the files in <*files*> that are present in the library into discrete files with the same names. If no <*files*> are specified, all files in the library are extracted.

At most one of the options **-[c r t x]** may be specified at the same time. If none of these is specified, the **-t** option is assumed.

Return Status

clib returns success if no problems are encountered. Otherwise it returns failure. After most failures, an error message is printed to STDERR and the library file is not modified. Output from the **-t**, **-s** options, and verbose remarks, are written to STDOUT.

Examples

To build a library and check its contents:

```
clib -c libc one.o two.o three.o
clib -t libc
```

will output:

```
one.o
two.o
three.o
```

To build a library from a list file:

```
clib -ci list libc six.o seven.o
```

where *list* contains:

```
# files for the libc library
one.o two.o three.o
four.o
five.o
```

The cobj Utility

You use **cobj** to inspect relocatable object files or executable. Such files may have been output by the assembler or by the linker. *cobj* can be used to check the size and configuration of relocatable object files or to output information from their symbol tables.

Command Line Options

cobj accepts the following options, each of which is described in detail below.

```
  cobj [options] file
        -d     output data flows
        -h     output header
        -n     output sections
        -o*    output file name
        -r     output relocation flows
        -s     output symbol table
        -v     display file addresses
        -x     output debug symbols
```

<*file*> specifies a file, which must be in relocatable format or executable format.

-d	output in hexadecimal the data part of each section.
-h	display all the fields of the object file header.
-n	display the name, size and attribute of each section.
-o*	write output module to file *. The default is STDOUT.
-r	output in symbolic form the relocation part of each section.
-s	display the symbol table.
-v	display seek addresses inside the object file.
-x	display the debug symbol table.

If none of these options is specified, the default is **-hns**.

Return Status

cobj returns success if no diagnostics are produced (*i.e.* if all reads are successful and all file formats are valid).

Examples

For example, to get the symbol table:

```
   cobj -s acia.o
```

```
symbols:

_main:    0000003e section .text defined public
_outch:   0000001b section .text defined public
_buffer:  00000000 section .bss defined public
_ptecr:   00000000 section .bsct defined public zpage
_getch:   00000000 section .text defined public
_ptlec:   00000002 section .bsct defined public zpage
_recept:  00000028 section .text defined public
```

The information for each symbol is: name, address, section to which it belongs and attribute.

The cv695 Utility

cv695 is the utility used to convert a file produced by the linker into an IEEE695 format file.

Command Line Options

cv695 accepts the following options, each of which is described in detail below.

```
  cv695 [options] file
        +V4     do not offset locals
        -d      display usage info
        +dpage  file uses data paging (HC12 only)
        -mod?   select compiler model
        +old    produce old format
        -o*     output file name
        +page#  define pagination (HC12 only)
        -rb     reverse bitfield (L to R)
        -v      be verbose
```

<file> specifies a file, which must be in executable format.

-V4 output information as per as *cv695* converter V4.x version. This flag is provided for compatibility with older version of *cv695* version. **DO NOT USE UNLESS SPECIFICALLY INSTRUCTION TO DO SO**.

-dpage output banked data addresses. **DO NOT USE THIS OPTION ON NON BANKED DATA APPLICATION. THIS FLAG IS CURRENTLY ONLY MEANINGFULL FOR THE MC68HC12.**

-d dump to the screen the interface information such as: frame coding, register coding, *e.g.* all the processor specific coding for IEEE (note: some of these codings have been chosen by COSMIC because no specifications exist for them in the current published standard).

THIS INFORMATION IS ONLY RELEVANT FOR WRITING A READER OF THE PRODUCED IEEE FORMAT.

-mod? where **?** is a character used to specify the compilation model selected for the file to be converted.

THIS FLAG IS CURRENTLY ONLY MEANINGFULL FOR THE MC68HC16.

This flag mimics the flag used with C. Acceptable values are:

c	for compact model
s	for short model
t	for tiny model
l	for large model

+old output old format for MRI.

-o* where ***** is a filename. ***** is used to specify the output file for *cv695*. By default, if **-o** is not specified, *cv695* send its output to the file whose name is obtained from the input file by replacing the filename extension with ".695".

+page# output addresses in paged mode where # specifies the page type:

0	for no paging.
1	for pages with PHYSICAL ADDRESSES
2	for pages with banked addresses `<page><offset_in_page>`

By default linear physical addresses are output.

THIS FLAG IS CURRENTLY ONLY MEANINGFULL FOR THE MC68HC12.

-rb reverse bitfield from left to right.

-v select verbose mode. *cv695* will display information about its activity.

Return Status

cv695 returns success if no problems are encountered. Otherwise it returns failure.

Examples

Under MS/DOS, the command could be:

```
cv695 C:\test\basic.h05
```

and will produce: **`C:\test\basic.695`**

and the following command:

```
cv695 -o file C:\test\basic.h05
```

will produce: **`file`**

Under UNIX, the command could be:

```
cv695 /test/basic.h05
```

and will produce: **`test/basic.695`**

The cvdwarf Utility

cvdwarf is the utility used to convert a file produced by the linker into an IELF/DWARF format file.

Command Line Options

cvdwarf accepts the following options, each of which is described in detail below.

```
  cvdwarf [options] file
        +page#  define pagination (HC12 only)
        -o*     output file name
        -loc    complex location description
        -rb     reverse bitfield (L to R)
        -v      be verbose
```

<*file*> specifies a file, which must be in executable format.

+page# output addresses in paged mode where # specifies the page type:

1	for banked code
2	for banked data
3	both (code and data)

By default the banked mode is disable.

THIS FLAG IS CURRENTLY ONLY MEANINGFULL FOR THE MC68HC12.

-o* where ***** is a filename. ***** is used to specify the output file for *cvdwarf*. By default, if **-o** is not specified, *cvdwarf* send its output to the file whose name is obtained from the input file by replacing the filename extension with "**.elf**".

-loc location lists are used in place of location expressions whenever the object whose location is being described can change location during its lifetime. THIS POSSIBILITY IS NOT SUPPORTED BY ALL DEBUGGERS.

-rb	reverse bitfield from left to right.
-v	select verbose mode. *cvdwarf* will display information about its activity.

Return Status

cvdwarf returns success if no problems are encountered. Otherwise it returns failure.

Examples

Under MS/DOS, the command could be:

```
cvdwarfC:\test\basic.h05
```

and will produce: **`C:\test\basic.elf`**

and the following command:

```
cvdwarf -o file C:\test\basic.h05
```

will produce: **`file`**

Under UNIX, the command could be:

```
cvdwarf /test/basic.h05
```

and will produce: **`test/basic.elf`**

CHAPTER

A

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 (cp6805) 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

_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 #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 #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 #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 value - alias definition is not a constant expression

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 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 macro invocation: <name> - a *#define* macro defined without arguments has been invoked with arguments

bad proto argument type - function prototype argument is declared without an explicit type

bad real constant - an invalid real constant has been specified

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 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

const assignment - a *const* object is specified as left operand of an assignment operator

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

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

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 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 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

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 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 in 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 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 substracted

invalid lvalue - the left operand of an assignment operator is not a variable or a pointer reference

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 substraction

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 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 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

unbalanced ' - a character constant does not end with a simple 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

zero divide - a divide by zero was detected

zero modulus - a modulus by zero was detected

Code Generator (cg6805) 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.

cannot call @interrupt function - an *@interrupt* function has been called directly. Its name can only be used in the interrupt vector table.

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 *cp6805* you used to compile your program is incompatible with *cg6805*.

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 (ca6805) 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.

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.

unknown directive - a directive name has not been recognized as a linker directive.

APPENDIX
B

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 **cx6805.cxf**.

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

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 **cx6805**.

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 **cx6805.cxf** as provided by the default installation appears below:

```
# CONFIGURATION FILE FOR 68HC05 COMPILER
# Copyright (c) 1996 by COSMIC Software
#
-pu                       # unsigned char
-pm0x1024                   # model configuration
-i c:\cx32\h6805          # include path
#
-m debug:x                # debug: produce debug info
-m nobss:,bss             # nobss: do not use bss
-m jmp:,,tjmptab.s        # jmp: optimize function call
-m nocst:,ct              # nocst: constant in text section
-m nsh:,nsh               # nsh: static not shared
-m rev:rb                 # rev: reverse bit field order
```

The following command line:

```
cx6805 hello.c
```

in combination with the above configuration file directs the **cx6805** compiler to execute the following commands:

```
cp6805 -o \2.cx1 -u -m0x1024 -i\cx32\h6805 hello.c
cg6805 -o \2.cx2 \2.cx1
co6805 -o \2.cx1 \2.cx2
ca6805 -o hello.o -i\cx32\h6805 \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**ssembler) 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:l,dl1,,,dl2      # dl1: line option 1
-m dl2:l,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

```
cx6805 +nobss +rev hello.c
```

in combination with the previous configuration file directs the **cx6805** compiler to execute the following commands:

```
cp6805 -o \2.cx1 -u -m0x1024 -rb -i\cx32\h6805 hello.c
cg6805 -o \2.cx2 -bss \2.cx1
co6805 -o \2.cx1 \2.cx2
ca6805-o hello.o -i\cx32\h6805 \2.cx1
```

APPENDIX
C

MC68HC05 Machine Library

This appendix describes each of the functions in the Machine Library (**libm**). These functions provide the interface between the MC68HC05 microcontroller hardware and the functions required by the code generator. They are described in reference form, and listed alphabetically.

Note that machine library functions handle values as follows:

- **integer** in a register pair, **ax**, **x** and the memory location **c_h**, or in the **a** register and the memory location **c_reg**. The **a** register always hold the less significant byte. The **x** register holds the most significant byte when used by the register pair **ax**, and holds the less significant byte when used by the register **x** and the memory location **c_h**.

- **longs** and **floats** in the four byte memory location **c_lreg**, ("float register" or "long register" depending on context).

- **pointer** to **long** or **float** in internal memory in **x**, and in **x** and the memory location **c_h** otherwise.

The library functions using a pointer to external memory (or code) have a name beginning with the ‘**x**’ letter, and the pointer is located in the pair composed by the **x** register for the lower byte, and the memory location **c_h** for the upper byte. The following describes only the function handling data in internal memory. Their equivalent functions have the same description except for the pointer location and size.

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 **cx6805** compiler uses to control the compilation process.

This appendix describes each of the passes of the compiler:

cp6805	the parser
cg6805	the code generator
co6805	the assembly language optimizer
ct6805	the jump table creator

The cp6805 Parser

cp6805 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 **cg6805**.

Command Line Options

cp6805 accepts the following options, each of which is described in detail below:

```
  cp6805 [options] file
          -ck   extra type checkings
          -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   do enum optimization
          -np   allow pointer narrowing
          -o*   output file name
          -p    need prototypes
          -rb   reverse bitfield order
          -sa   strict ANSI conformance
          -u    plain char is unsigned
          -xd   debug info for data
          -xp   no path in debug info
          -xx   extended debug info
          -x    output debug info
```

-ck direct the compiler to enforce stronger type checking.

-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. *cp6805* 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.
-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.

-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.

-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

cp6805 returns success if it produces no error diagnostics.

Example

cp6805 is usually invoked before *cg6805* the code generator, as in:

```
cp6805 -o \2.cx1 -u -i \cx32\h6805 file.c
cg6805 -o \2.cx2 \2.cx1
```

The cg6805 Code Generator

cg6805 is the code generating pass of the C compiler. It accepts a sequential file of flow graphs and parse trees from *cp6805* 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

cg6805 accepts the following options, each of which is described in detail below:

```
  cg6805 [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
         -no     do not use optimizer
         -nsh    do not share locals
         -o*     output file name
         -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*.
-na	do not produce an *xdef* directive for the *equate* names created for each C object declared with an absolute address.
-no	do not produce special directives for the post-optimizer.
-nsh	do not share memory areas allocated for function local areas.
-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	When this option is set, each function name is send to STDERR when *cg6805* starts processing it.

Return Status

cg6805 returns success if it produces no diagnostics.

Example

cg6805 usually follows *cp6805* as follows:

```
cp6805 -o \2.cx1 -u -i\cx\h6805 file.c
cg6805 -o \2.cx2 \2.cx1
```

The co6805 Assembly Language Optimizer

co6805 is the code optimizing pass of the C compiler. It reads source files of MC68HC05 assembly language source code, as generated by the *cg6805* code generator, and writes assembly language statements. *co6805* is a peephole optimizer; it works by checking lines function by function for specific patterns. If the patterns are present, *co6805* 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

co6805 accepts the following options, each of which is described in detail below:

```
 co6805 [options] <file>
       -c      keep original lines as comments
       -d*     disable specific optimizations
       -o*     output file name
       -t*     table 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.
-t*	use the jump table *****, allowing the optimizer to perform automatically the function name replacement.

NOTE

The current implementation requires the jump table file name to be: ***jmptab.s****.*

-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

co6805 returns success if it produces no diagnostics.

Example

co6805 is usually invoked after *cg6805* as follows:

```
cp6805 -o \2.cx1 -u -i\cx\h6805 file.c
cg6805 -o \2.cx2 \2.cx1
co6805 -o file.s \2.cx2
```

The ct6805 Utility

The *ct6805* utility reads the executable file produced by the linker and finds the name of all the object files of the application. *ct6805*then scans all the listing files, if any, and creates as output an assembly source file containing a replacement label followed by a jump instruction to the target function, for each of the selected function. The selected function list is built by extracting the sixteen most used function names following a *jp* instruction, including the library functions.

Command Line Options

ct6805 accepts the following options, each of which is described in detail below:

```
ct6805 [options] <file>
       -n#     maximum number of functions
       -o*     output file name
       -v      echo processed file names
```

-n#	maximum number of functions output in the jump table. Default value is 16, producing a 48 bytes wide area.
-o*	writes the jump table to the file *****.
-v	writes the name of the scanned files on the terminal screen.

Return Status

ct6805 returns success if it produces no diagnostics.

Example

For example, from the *acia.h05* file built by the *test* program provided with the package, run the following command:

```
ct6805 -o jmptab.s acia.h05
```

This command produces the following result in the *jmptab.s* file:

```
;      JUMP TABLE FOR 68HC05
;      Copyright (c) 1995 by COSMIC Software
;
R_main:
      jmp   _main
R_outch:
      jmp   _outch
R_getch:
      jmp   _getch
;
      xdef  R_main
      xref  _main
      xdef  R_outch
      xref  _outch
      xdef  R_getch
      xref  _getch
      end
```

Index