CHAPTER 1

INTRODUCTION

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

Introduction

The C cross compiler targeting the Z80/HD64180 is an integrated series of software development tools. It reads C source files, assembly language source files, and object code files, and produces as output 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 **cxdb,** the C source level cross debugger or by other debuggers or in-circuit emulators.

You begin compilation by invoking the **c** compiler driver. **c** then opens a command file, called a "prototype file." **c** executes the commands in the prototype file sequentially. The options you specify when you invoke **c** determine the specific combinations of options that it passes to the programs that make up the compilation process. If you plan to use the **cxdb** C source level cross debugger to debug the program you are compiling, for example, you must specify the **-dxdebug** option to **c.**

Support for ROMable Code

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

o Referencing of absolute hardware addresses;

- Control of the Z80/HD64180 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 your target system.

Optimizations

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

- The compiler uses register **hl** to hold the 16 least significant bits of the first data object passed on a function call
- The compiler will perform arithmetic operations in character precision if the types of the operands are 8-bit.
- Branch shortening logic chooses the smallest possible jump/branch instructions. Jumps to jumps and jumps over jumps are eliminated as well.
- Integer constant expressions are folded at compile time.
- An optimized **switch** statement produces combinations of jump tables for closely spaced **case** labels, a scan table for a small group of loosely spaced **case** labels, or a sorted table for a binary search.
- The compiler eliminates unreachable code.
- You can use optional strict single precision floating point arithmetic.
- Redundant load and store operations are removed.
- The functions in the C library are packaged in three separate libraries; one is made up of functions that do not require 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 versions of the modules needed.

Others are single precision and double precision versions of libraries.

Compiler Architecture

The C compiler consists of several programs that work together to translate your C source files to executable files and listings. **c** controls the operation of these programs automatically, using the information you provide in the options you specify to it in combination with the contents of the prototype file.

c runs the programs described below in the order listed.

cpp80 - the preprocessing pass of the C compiler. **cpp80** expands directives in your C source before actual compilation begins.

cp180 - the language parsing pass of the C compiler. **cp180** accepts the output of **cpp80** and processes it for input to the code generator.

cp280 - the code generating pass of the C compiler. **cp280** accepts the output of **cp180** and generates an intermediate language

cp380 - the optimizer. **opt80** optimizes the language that **cp280** generates, and produces assembly language.

x80 - the assembler. **x80** converts the assembly language output of **cp380** to a relocatable object module.

lnk80 - the linker. **lnk80** combines object modules to produce an executable image.

Programming Support Utilities

Once **c** has run the compiler, assembler, and linker to produce an executable image for your target system, you may use the programming support utilities listed below to inspect and reconfigure the executable.

toprom - automatic data initializator. **toprom** modifies an executable image produced by the linker to move all initialized data into a code section which will be programmed into a PROM. The startup program will copy these data into RAM before to start the application.

hex80 - absolute hex file generator. **hex80** translates executable images produced by the linker into any one of several hexadecimal interchange formats, for use with in-circuit emulators and PROM programmers. **hex80** produces the following formats:

standard Intel hex format

Motorola S-record format

Tektronix standard hex format

Tektronix Extended hex format

hex80 also allows you to change the bias of the text and data section load addresses.

unhex - hexadecimal file format translator. **unhex** is a universal hex file translator. It allows you to convert from Intel or Motorola hex formats back to an absolute xeq hex format file. This helps you to translate binary files to and from a text form that can be sent more safely over nontransparent data links.

rel80 - object module inspector. **rel80** allows you to examine libraries and relocatable object files for symbol table information and to determine their size and configuration.

lby - build and maintain object module libraries. **lby** 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.

lord80 - object module library ordering utility. **lord80** creates a list of module names such that, where possible, no library module depends on an earlier module in the list. **lby** uses this information to create a library file that contains no circular dependencies. This decreases link time.

Also included are the programs **lm** and **pr,** which are used to produce listings as described below.

You may then use the programming support utilities **rel80, hex80, unhex, toprom, lby** and **lord80** to inspect and reconfigure the executable image.

Linking

Unless you specify otherwise, **c** runs the linker to combine 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 **hex80** utility, can be downloaded and run on your target system. If you specify the debugging option when you invoke **c**, the compiler driver will generate a Version 3.3 object module, which contains debugging information. You can then use **cxdb**, the C source debugger to debug your code on your host system, or use the **lines** and **prdb** utilities to extract the debugging information for use by another debugger or in-circuit emulator.

Listings

You have several options for obtaining listings. If you request no listings, then **c** error messages from each compiler pass are directed to your terminal, but no additional information is provided. Each error is labelled with the C source file and line number where the compiler detected the error.

If you request a C source listing interspersed with error messages, the compiler merges the error messages it generates as comments below the lines of C source at which it detected the errors. Unless you specify otherwise, the error messages are still written to your terminal as well. **c** runs a separate utility to produce the combined file. Your listing is the output of this utility, which is called **lm**.

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. Error messages also become comments. Unless you specify otherwise, the error messages are still written to your terminal as well. **c** runs the **lm** utility to produce the combined file which is then used as input to the assembler. Your listing is the listing output from the assembler.

If you request a listing of assembly language output without C source lines or object code, **c** gathers error messages from the three compiler passes and merges them with the assembly language output of the code generator to produce the listing. Unless you specify otherwise, the error messages are still written to your terminal as well. The **lm** utility for merging the listing is used, but the combined file serves as input to a separate page formatting utility instead of the assembler. Your listing is the output from the page formatting utility, which is called **pr**.

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 support utilities, see Chapter 8. For information on the compiler passes, see Appendix D.

Host System Requirements

To install and operate the C cross compiler on your host system, you must provide the following:

- A text editor that operates on ASCII text files. Only source code programs that are ASCII text files are suitable for processing by the C compiler.
- A means of communication between your host system and your target system, such as a direct serial connection. You must be able to download executable images to your target system.
- For 8086 based systems, you must have a hard disk to use the compiler and at least one floppy disk drive to install the compiler.
- For 8086 based systems, you must have a minimum of 128K bytes of memory to use the compiler.

CHAPTER 2

TUTORIAL INTRODUCTION

This chapter shows you step by step how to compile, assemble and link the example program **acia.c**, which is included on your distribution media. The following is a listing of **acia.c**:

```
/*
 *
 *  Each received character is copied in a buffer by
 *      by the interrupt routine. The main program reads
 *      characters from the buffer and echoes them.
 *
 */
#define SIZE    512 /* size of the buffer */

/* the input port is a char at port 2
 * the output port is a char at port 5
 */
@port char input @0x02;
@port char output @0x05;

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

/* "builtin" defined function for enabling interrupts
 */
@builtin ei() @0xfb;

/*  character reception. loops until a character is received
 */
char getch()
    {
    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 output port
 */
outch(c)
    char c;
    {
    output = c;         /* send it */
    }

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

/*  Main program : an infinite loop
 *  of receive transmit
 */
main()
    {
    char c;                 /* received char */

    ptecr = ptlec = buffer;         /* initialize pointers */
    ei();                   /*enable interrupts */
    for (;;)                /* loop */
        {
        c = getch();            /* get a char */
        outch(c);           /* send it */
        }
    }
```

Default Compiler Operation

By default, the c compiler driver compiles, assembles and links your program, creating an object module named **xeq.80** in standard object format.

The example below shows what you see on your screen if you invoke the compiler driver without specifying any command line options.

As it processes the command line, c echoes the name of each input file to the standard output file (your terminal screen by default). When it runs the linker, it echos the name of the object file it creates to your terminal screen also. You can change the amount of information the compiler driver sends to your terminal screen using command line options, as described below.

To compile, assemble and link **acia.c** using default information, enter the command:

```
    c acia.c
```

The compiler driver writes the name of the input file it processes and the name of the output file it creates to your terminal screen:

```
    acia.c:
    xeq.80:
```

The result of the compilation process is an object module named **xeq.80** in standard object format. **xeq.80** can now be processed by the **hex80** utility to translate it to Motorola standard S-record format. You can then download the reformatted object to your target system.

Specifying Command Line Options

You specify command line options directly to **c** to control the compilation process. For instance, you may want to see more information about the programs **c** runs and the options it passes to them. You may want to obtain a listing of the assembly language code that the compiler generates. You may also want to change the default name that **c** gives the object module output, so that you do not overwrite an existing object module named **xeq.80.**

The **-v** option directs the compiler driver to echo the name of each program it runs and the options it passes to each program to your terminal screen. The **-dlistcs** option directs the compiler driver to create a mixed section listing of C code and assembly language code in the file **acia.ls.** The **-o** option directs the compiler driver to name the object module that the linker creates "**testshot.**" See Chapter 4 for information on these and other command line options.

To perform the operations described above, enter the command:

```
    c -v -dlistcs -otestshot acia.c
```

The compiler driver writes to your terminal screen something similar to the following:

```
prototype file: c.pro
acia.c:
cpp80 -o c4d.c0 -x -err acia.c
cp180 -o c4d.c1 -mu -err c4d.c0
cp280 -o c4d.c0 +list -err c4d.c1
cp380 -o c4d.c1 -e -r30 c4d.c0
lm -o acia.tmp -err -lt -c; c4d.c1
del c4d.c1
ren acia.tmp c4d.c1
x80 -o acia.o +l c4.c1 > acia.ls
lnk80 - < xeq.lnk
del xeq.lnk
```

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

o the C source file **acia.c**

o the C and assembly language listing **acia.ls**

o the object module **acia.o**

o the executable file **testshot**

Object Module Format Translation

Although **testshot** is an executable image, it is not yet in the correct format to execute on your target system. You use the **hex80** utility to translate the format of executables. To translate the format of **testshot** to Intel hex format, enter the command:

```
hex80 testshot > test
```

test is now an executable image in Intel hex format, and is ready to be loaded on your target system. Open a serial port and download the executable, or use the **cxdb** C source level cross debugger to debug and test the program on your host system.