CHAPTER 3

PROGRAMMING FOR Z80/HD64180 ENVIRONMENTS

This chapter explains how to perform special tasks that many applications require.

Modifying the Runtime Startup

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

- Initializes the bss section if a bss section is needed,
- Copies ROM into RAM if required,
- Initializes the stack pointer,
- Calls **_main** or other program entry point, and
- Provides 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.80** included on your distribution media. It does not performs automatic data initialization. A special startup program is provided, **crtsrom.80,** which is used instead of **crts.80** when the **-dprom** option is specified on the compiler command line. It is used in conjunction with the **toprom** utility, which moves the initialized data into code section. If you change the name of the runtime startup file you must edit your prototype file as described in Appendix B. The runtime startup file can be placed anywhere in the memory map. It easier to locate if it is first on the link-line, but that is not mandatory. Usually, when the user application is running, the program startup will be "linked" with the RESET interrupt, and the startup file may be at any convenient location.

```
1;
2;
3;          SAMPLE STARTUP CODE FOR FREESTANDING SYSTEM
4;          Copyright (c) 1989 by COSMIC (France)
5;
6           .external _main
7           .external __memory
8           .public _exit
9           .public __text
10          .public __data
11          .public __bss
12;
13;
14;         PROGRAM STARTS HERE SINCE THIS FILE IS LINKED FIRST
15;
16;         First we must zero bss if needed
17;
18          .psect _text
19__text:
20          ld hl, __memory ; __memory is the end of the bss
21             ; it is defined by the link line
22          ld de, __bss ; __bss is the start of bss (see below)
23          sub a
24          sbc hl, de  ; compute size of bss
25          jr z, bssok ; if zero do nothing
26          ex de, hl
27loop:
28          ld (hl), 0  ; zero bss
29          inc hl
30          dec de
31          ld a, e
32          or d
33          jr nz, loop ; any more left ???
34bssok:
35;
36;         Then set up stack
37;
38;         The code below sets up an 8K byte stack
39;
40;         after the bss. This code can be modified
41;
42;         to set up stack in any other convenient way
43;
44          ld bc,  __memory ; get end of bss
45          ld ix, 8192 ; ix = 8K
46          add ix, bc  ; ix = end of mem + 8k
47          ld sp, ix  ; init sp
48;
49;
50;
51;         Then call main
52;
53          call _main
54_exit:     ; exit code
```

```
55          jr _exit  ; for now loop
56;
57;
58;
59          .psect _data
60__data:
61          .byte 0 ; NULL cannot be a valid pointer
62;
63;
64;
65          .psect _bss
66__bss:     ; define start of bss
67          .end
```

Description of Runtime Startup Code

_main is the entry point into the user C program. **__memory** is an external symbol set by the linker at the end of the bss section.

The start of the bss section is marked by the local symbol **__sbss,**

Lines **19** to **33** reset the bss section.

Lines **44** to **47** compute and set the stack pointer. **By default, the startup provides a 8K stack directly above the bss section. You may have to modify this portion of the module to meet the needs of your application.**

Line **53** calls the **main()** routine in the user's C program.

Lines **54** to **55** trap a return from the user's **main()** routine. **If your application must return to a monitor, for example, you must modify this portion of the module.**

Initializing data in RAM

If you have initialized static variables, which are located in RAM memory, you need to do the static initialization before to start the C program. The linker provides no way of doing it; the utility **toprom** must be used to that effect. It moves all initialized data segments after the first code segment, and creates a descriptor giving the starting address, destination and size for each segment.

To use this table, and copy data in ram, you need to locate the table in the prom. This is done by defining a

symbol at link time, which specifies the place where the table will be created by **toprom**. Then, you have to copy the data from PROM to RAM, using information obtained from the descriptor created by **toprom**.

All these manipulations are done automatically when you set the **-dprom** option on the compiler command line. Refer to Chapter 8, **toprom** utility, for information about the descriptor format.

Performing Input/Output in C

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

The C source code for these and all other C library functions is included with the distribution, so that you can modify them to meet your specific needs. Note that all input/output performed by C library functions is supported by underlying calls to **getchar** and **putchar**. These two functions provide access to all input/output library functions. The library is built this 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();
char putchar(char c)
char c;
```

Referencing Absolute Addresses

The 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 reference. You can give a memory location a symbolic name and associated type, and use it as you would any C identifier. This feature is useful for accessing memory or for calling functions at a known address 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 a byte at address 0x40 with the identifier name **ttystat**, write a definition of the form:

```
char ttystat @0x40;
```

where @0x40 is an absolute address and not a data initializer.

To use the byte at 0x40 in your application, write:

```
char c;

c = ttystat;     /* to read byte */
ttystat = c;     /* to write byte */
```

Similarly, to associate the absolute address 0x2000 with the entry point of the function **printer_cntrl**, write a function declaration of the form:

```
int printer_cntrl() @0x2000;
```

You may then invoke the function **printer_cntrl()** in the same manner as any other C function.

The const and volatile Type Qualifiers

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

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 data object by the function call. 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 indeterminate.

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.

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 changed */
```

For more information on type qualifiers, refer to Chapter 2 of your C Language Specification for Microcontroller Environments.

Bank Switching

Bank switching support for the **HD64180** has been added for large code section programs, and is accessed through use of the **@far** extension. The **@far** extension is used to specify that a function is in another bank. For example to specify that function **foo** which returns a char will be in another bank, one would write:

```
@far char foo();
```

this will direct the compiler to output the proper code needed to call **foo**.

Support has been added in such a way to make it transparent for the called function whether it has been called from another bank or not (i.e. was declared as a **far** function by the caller).

Only the **caller** of a function, that is in another bank, **need declare** that same function as a "**far** function".

This makes it much easier to create and maintain libraries since modules can be called from the same bank or from another bank depending on your own requirements.

It also relieves the user from maintaining two different set of libraries composed with the same modules, one which would be for bank-switching calls, the other one being for standard calls.

"**far** calls" are supported through the use of the **IY** register.

If a "**far** call" is present in a module, space is allocated on the stack (and is pointed at by register **IY)** to allow for the "**far** call" (that is to save the return address and the bank number). When a "**far** call" is made the compiler will "redirect" the call to a call of the **c.libc** routine (which is described in Appendix C.) with the address of a **"far function pointer"** in register **BC. c.libc** saves the proper information then calls the function really called.

The called function **must return** to **c.libc** which will in turn restore the appropriate information and will return to caller.

This is an example of how to make "**far** calls" and build the proper executable file:

```
file main.c:
============


@far int func1();   /* say it is a far call */
@far void func2();  /* say it is a far call */

main(a, b, c)
    int a;
    char b, c;
    {

    func1(1, 2, 3);
    func2();
    }


file func1.c:
=============

int func1(i, j, k)
    int i, j, k;
    {
    return (i ? j + k : j - k);
```

```
}
```

```
file func2.c:
==============

extern int flag;

void func2()
    {

    ++flag;
    }
```

To link the program the link command should be:

```
-ps12 +h
+text -f12 -b 0x2000 -o0x2000 func1.o
+text -o0x2000 func2.o
+text -b0x1000 -f0 -o0x1000 main.o libc.180 libm.180
```

All of the flags used in the link command are fully documented in the chapter relative to the linker (cf. chapter 6). This is a quick explanation of the above example.

-ps12 is used to indicate that the size of the banks is 2**12 i.e. **4K.**

+h is used to specify a multi segment output file; this flag is mandatory when using **bank-switching.**

+text -f12 -b 0x2000 -o0x2000 called1.o means that we start a new segment of code which will be filled to a **4K** size **(-f12),** and whose physical and logical addresses will both be **0x2000.** This segment (in fact a bank) will hold function func1().

+text -o0x2000 func2.o means the we start a new segment whose logical address is **0x2000 (-o0x2000);** its physical address will be **0x3000** since previous bank was at physical address **0x2000** and was completed to **4K** size. This segment (or bank) will contain function func2().

+text -b0x1000 -f0 -o0x1000 main.o libc.180 libm.180 means that we start a new segment whose physical and logical addresses are **0x1000 (-b0x1000** and **-o0x1000),** and which will contain main() and the libraries. (Note also that this segment will not be completed to **4k** since the "filler" has been reset to zero **(-f0)).**

Using the **bank-switching** requires that you carefully read the section devoted to the linker **lnk80.**

Placing Data Objects in the bss Section

The compiler automatically places uninitialized static data objects in the bss section. All such data objects are published via a **.external _bss** declaration on the assembly language level. All initialized static data objects are placed in the data section. At link time, the bss section is placed at the end of the data section.

Generating Inline Machine Instructions; the @builtin function

Calling an assembly language routine from within a C program can decrease performance due to the overhead associated with the call. For this reason Whitesmiths C provides the type qualifier **@builtin.** Functions you declare with the **@builtin** type qualifier generate inline machine instructions when referenced.

When a **@builtin** function is called, the compiler generates an inline constant corresponding to the value or absolute address of the function being referenced. The size of the constant is one to four bytes. The actual machine code, not the mnemonic, is generated. For the C user, this means access to the reserved, privileged, or otherwise inaccessible instructions of the processor. This is also the only way (from C) to produce instruction codes that are unimplemented in the assembler.

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

```
@builtin void scf(void) @0x37;
```

You must specify an absolute location for a builtin function. When you call a builtin function within an expression, the compiler generates up to four bytes of code in the instruction stream. The value it generates is made up of the lower bytes of the absolute location, without any leading **nuls.**

Inserting Inline Assembly Instructions: the _asm() function

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

_asm(<string constant>)

The string constant arguments are the assembly code you want embedded in your C program. **The strings you specify follow standard C rules. For example, carriage returns can be denoted by the '\n' character.** For example, to produce the following asembly sequence:

```
ld  hl, (memory)
call    _func
you would write

_asm(" ld hl, (memory)\n call _func\n");
```

The argument string must be shorter than 512 characters. If you wish to insert longer assembly code strings you will have to split your input among consecutive calls to **_asm()**.

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

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

```
    if (_asm("ld b,0\nld a,i\nld c, a\n") == 0x10)
```

allows to test the value of the interrupt register I. That way, you can use **_asm()** to write equivalents of C functions directly in assembly language.

Generating ei, di, reti and halt Instructions

You can use the **@builtin** type qualifier to generate **di, ei, reti,** and **halt** instructions from C. First you define the appropriate functions, which will generate assembly language code when your program calls them.

```
@builtin di() @0xf3;
@builtin ei() @0xfb;
@builtin reti() @0xed4d;
@builtin halt() @0x76;
```

To generate these instructions in your program, write:

```
di();
ei();
reti();
halt();
```

The example program **acia.c** uses this method.

Writing Interrupt Handlers

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

When you define an **@port** function, the compiler selects a special "rti" instruction for the return sequence.

You define an **@port** function in one of two ways. You define it for use in expressions by using the type qualifier **@port** to qualify the type returned by a function you declare. An example of such a definition is:

```
@port void syscall(void) @0X2000;
```

For this form, you must specify an absolute location for a **@port** function.

The second way to define a **@port** function is by writing a normal function body in place of the absolute location as in

```
@port void sys_handler(int type_code)
    {
    ...
    }
```

When you call an **@port** function from a C function, the compiler generates an entry sequence that saves all registers. You use this form to handle exceptions and interrupts.

Both **@port** functions and **@builtin** functions are extensions to the proposed ANSI standard.

C Interface to Assembly Language

The C cross compiler translates C programs to run on the Z80/HD64180 into assembly language according to the specifications described in this section.

You may write external identifiers in both uppercase and lowercase. The first 127 letters must be distinct. The compiler prepends an underscore '_' character to each identifier.

By default, the compiler places function text in the _text section. Function text is not to be altered or read as data. External function names are published via **.public** declarations.

Literal data such as strings, double constants, and switch tables, are normally generated into the **_text** section.

By default, the compiler generates initialized data into the **_data** section. External data names are published via **.public** declarations. Data you declare to be of "const" type by adding the type qualifier **const** to its base type is generated into the **_text** section. Data initialized to zero is generated into the **_bss** section.

Uninitialized declarations result in an **.external** reference, one instance per program file.

Function calls are performed according to the following three steps:

1. Arguments, **except the first one,** are moved onto the stack from right to left. **The first argument** is stored in the **hl** register if its size is less than or equal to the size of an int; if the first argument is larger than 16 bits, the **hl** register will hold the least significant 16 bits, while the most significant part of the argument will be pushed on the stack.

2. The function is called via a **call _func** instruction.

3. The arguments to the function are popped off the stack.

4. A data space address is moved onto the stack if a struct area is required.

Register Usage

The C compiler uses **in-core pseude registers.** These memory areas are used to handle floating point, **long** variables and to simulate the **register** storage class. These **pseudo registers** are: **c.r0, c.r1, c.r2** and **c.r3. c.r0** and **c.r1** are each eight-byte memory areas that are not preserved across calls. **c.r2** and **c.r3** are each two-byte memory areas that must be preserved across calls.

Except for the return value, the registers **af, bc, hl, and the in-core pseudo registers c.r0, c.r1,** and the condition codes are undefined on return from a function call. Register **de** is used to hold **C register variables;** it is preserved across calls. Register **iy** is used for bank swit-

ching on the HD64180; it should be preserved across calls, if the **@far** extension is used. The return value is in **bc** if it is of type char, short, integer or pointer to.... The return value is in the pseudo register **c.r0** if it is of type long or float.

Stack frames are maintained by each C function, using **ix** as a frame pointer. On entry to a function, a call to one of: **c.sav, c.sav0, c.savs** or **c.savs0** will stack **ix** and leave it pointing at the stacked **ix; c.sav** and **c.sav0** will also stack **de, c.r2** and **c.r3**. Arguments starts at **4(ix)**; auto storage may be reserved on the stack at **-7(ix)**. To return a jump to one of: **c.ret, c.ret0, c.rets** or **c.rets0** (according to earlier call) will use **ix** to restore **ix, sp**, and **de, c.r2** and **c.r3** if necessary, the return.

Data Representation

Data objects of type **short int** and **int** are stored as two bytes, less significant byte first. Data objects of type **long integer** are stored as two short integers, more significant integer first. Data object of type **double** are stored as four integers (eight bytes), most significant integer first, i.e., in the order (6, 7, 4, 5, 2, 3, , 0, 1). Representation is the same as for PDP-11 computers: most significant bit is one for negative numbers, else zero; next eight bits are the characteristic, biased such that the binary exponent of the number is the characteristic minus 0200; remaining bits are the fraction, starting with the 1/4 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 make shortcuts. Otherwise, there is an assumed 1/2 added to all fractions to put them in the interval [0.5, 1.[. The value of the number is the fraction, times -1 if the sign bit is set, times two raised to the exponent. Data object of type **float** are stored as two integers, most significant integer first. Representation is same as double, with the last four bytes discarded,i.e. the four least significant fraction bytes.