HI-TECH SOFTWARE
C COMPILER (Z80)

User's Manual

October 1989

CONTENTS

1. Compiler Installation

The following information relates primarily to the cross compiler version of the Z80 compiler. See the end of this chapter for information relating to the CP/M native version of the compiler. Note that even the cross compiler produces by default CP/M .COM files. To generate HEX or binary code for a ROM system see the section "Run-Time Environment".

1.1. Hardware Requirements

HI-TECH C runs on MS-DOS machines with at least 256K of memory and a hard disk.

1.2. Installation

HI-TECH C is normally supplied on 2 floppy disks. There is an install program on disk #1 which will install the compiler onto your hard disk. It prompts you for any necessary information. The compiler is normally installed into a directory called HITECH on your hard disk.

1.3. .HUF Files

The source for the run-time library routines is supplied in four .HUF files. These will be found in the SOURCES directory of the second disk. They are:

GEN.HUF	Non-I/O routines
STDIO.HUF	Standard I/O routines
CPM.HUF	Low-level I/O routines
FLOAT.HUF	Floating point support routines

The first three all go to make up ZLIBC.LIB, the last ZLIBF.LIB. CPM.HUF also contains the source for the start-off code, ZCRTCPM.OBJ and ZRRTCPM.OBJ. Each .HUF file contains several C and assembler source files compressed into a Huffman encoded format. The DEHUFF utility should be used to examine or extract the contents. The DEHUFF utility is used as follows:

To list the contents of a .HUF file

```
DEHUFF FILE.HUF
```

To extract all files from a .HUF file

```
DEHUFF -X FILE.HUF
```

To extract only named files from a .HUF file

```
DEHUFF -X FILE.HUF AFILE.C ANOTHER.C ...
```

In these examples, you should substitute the name of the .HUF file you are interested in, e.g. STDIO.HUF. The names specified in the last example to be extracted should match file names listed as contained in that .HUF file.

1.4. What Went Wrong

There are numerous error messages that the compiler may produce. Most of these relate to errors in the source code (syntax errors of various kinds and so forth) but some represent limitations, particularly of memory. The two passes most likely to be affected by memory limitations are CGEN and OPTIM. CGEN will issue the message "No room" if it runs out of dynamic memory. This can usually be eliminated by simplifying the expression at the line nominated in the error message. The more complex the expression, the more memory is required to store the tree representing it. Reducing the number of symbols used in the program will also help.

Note that this error is different from the message "Expression too complicated" which indicates that the expression is in some way too difficult for the code generator to handle. This message will be encountered very infrequently, and can be eliminated by changing the expression in some way, e.g. computing an intermediate value into a temporary variable.

The optimizer OPTIM reads the assembler code for a whole function into memory at one time. Very large functions will not fit, giving the error message "Optim: out of memory in _func" where func is the name of the function responsible. In this case the function should be broken up into smaller functions. This will only occur with functions with several hundred lines of C source code. Good coding practice will normally limit functions to less than 50 lines each.

If you get a strange error from P1, CGEN, ZAS or OPTIM it is worth checking that there is sufficient disk space on whichever drive you are working on. Normally if disk space is used up a pass will exit with the message "Error closing output file" or "Write error on file".

If you use a wordprocessing editor such as Wordstar, ensure that you use the "non-document" mode or whatever the corresponding mode is. The edited file should not contain any characters with the high bit set, and the line feed at the end of the line must be present. Lines should be not more than 255 characters long.

1.5. CP/M Version

The CP/M native compiler is frozen at V3.09. It does not have some features, especially interrupt and port type qualifiers. The compiler is distributed on two or more disks, depending on format. You should copy these disks to working disks as follows:

Desirably, you should have the following files on one disk:

```
STDIO.H
C.COM
$EXEC.COM
CPP.COM
P1.COM
CGEN.COM
OPTIM.COM
ZAS.COM
LINK.COM
CRTCPM.OBJ
LIBC.LIB
LIBF.LIB
DEBUG.COM
CREF.COM
OBJTOHEX.COM
LIBR.COM
RRTCPM.OBJ
CTYPE.H
CPM.H
SIGNAL.H
```

If you cannot fit all these on one disk, it is useful to have at least the files up to and including LIBC.LIB on a single disk. If there is still too much to go on one disk, you will need a two-drive (or more) system. In this case, you should put as many as possible of the files listed above onto one disk, then the remainder on a second disk. If your disk drives are very small, you should spread them out over as many disks as necessary. For example, on a Kaypro or Osborne with approx. 180k per disk, you can set up two disks thus:

Disk #1:

```
STDIO.H
C.COM
$EXEC.COM
CPP.COM
P1.COM
CGEN.COM
OPTIM.COM
```

Disk #2:

```
ZAS.COM
LINK.COM
LIBC.LIB
OBJTOHEX.COM
LIBF.LIB
LIBR.COM
DEBUG.COM
```

To compile a program, you should start with disk #1 in drive A: (if there is room, having your favourite editor on this disk is handy). Keep your source files on a disk in drive B:, and use B: as your current drive. Suppose you have created a file called FRED.C and wish to compile it to produce FRED.COM. With B: as the current drive, issue the command

```
A:C -V FRED.C
```

The -V flag means verbose; this is not necessary but is useful. You will then see on the screen messages indicating the progress of the compilation. When it reaches a stage which requires the presence of a compiler pass that is not on the disk in A: (at this stage disk #1) the message

```
Not found: change disks, hit a key
```

will appear. At this point, you should remove disk #1 from drive

A: and insert disk #2, then hit any key on the keyboard. The compilation will then resume.

If the compiler is spread over 3 disks, a similar message will be issued when the next disk change is required. Since your current drive is B:, all temporary files and the output files will all be created on B:. This is the key to the whole process, allowing the disk in A: to be changed at the appropriate points.

Remember that in general, the C command will only invoke the passes needed to produce a program from whatever input you give it. If you give it assembler files, it will assemble and link them. If you give it object files, it will just link them. At any point, if a required pass is not found on the disk in A:, you will be prompted to change disks. The only major limitation on this is that LINK must be on the same disk as the library files.

1.6. Reconfiguring

The compiler configuration is controlled by the C.COM program. It is supplied set up to expect the compiler files on the A: drive, under user number 0. This applies to the individual passes, header files and library files. Any or all of these may be put on another drive or under another user number by creating a file on your working disk called ENVIRON and placing in it suitable definitions for the compiler driver to use. The compiler driver will look for a line of the form

```
HITECH=C:
```

in the file. In this example, the compiler passes would be stored on drive C:. You may also specify a user number, e.g.

```
HITECH=1:D:
```

It is also possible to specify where temporary files should be created, e.g. if you have a ram disk as drive E: and you wish the compiler to create temporary files there (this speeds up compilation considerably) you would add to ENVIRON a line like

```
TEMP=E:
```

Do not use user numbers in a TEMP specification - this will confuse the compiler.

2. Runtime Organization

The runtime environment of HI-TECH C is quite straightforward; this is to a large degree due to the C language itself. C does not rely on a large runtime support module, rather there are a few library routines which are basic to the language, while all other runtime support is via specific library packages, e.g. the STDIO library.

2.1. Runtime Startup

The starting address of a C program is always the lowest code address; for CP/M this is the base of the TPA, 100H. The global symbol start is at this address. The code located at the start address performs some initialization, notably clearing of the bss (uninitialized data) psect. In the case of CP/M it also calls a routine to set up the argv, argc values. Depending on compile-time options, this routine may or may not expand wild cards in file names (? and *) and perform I/O redirection.

Having set up the argument list, the startup code calls the function _main. Note the underscore ('_') prepended to the function name. All symbols derived from external names in a C program have this character prepended by the compiler. This helps prevent conflict with assembler names. The function main() is, by definition of the C language, the "main program".

2.2. Stack Frame Organization

On entry to _main, and all other C functions, some code is executed to set up the local stack frame. This involves saving non-temporary registers (ix and iy for the Z80) and setting up the base pointer (ix) to point to the base of the new stack frame. Ix and iy are saved only if the function uses them. Then the stack pointer is adjusted to allow the necessary space for local variables. Typical code on entry to a function would be:

```
  push    ix
  ld      ix,0
  add     ix,sp
  ld      hl,-10
  add     hl,sp
  ld      sp,hl
  push    iy
```

This will allocate 10 bytes of stack space for local variables. The stack frame after this code will look something like this:

```
|     .      |
|     .      |
| arguments  |
|     to     |
|  function  |
|------------|
|  return    |
|  address   |
|------------|
|   saved    |
|    ix      |       <--- ix points here
|------------|
|   local    |
|   data     |
|     .      |
|     .      |
|     .      |
|            |
|------------|
|   saved    |
|    iy      |       <--- sp points here
*------------*
```

All references to the local data or parameters are made via ix. The first argument is located at (ix+4). A parameter may occupy any number of bytes, depending on its type. If a char is passed as an argument to a non-prototyped function, it is expanded to int length. Where a function is prototyped, a char argument will occupy only one byte.

Local variables are accessed at locations (ix-1) downwards. If there is a register variable (there is at most one) it will be in register iy. The alternate register set is used for removing parameters from the stack after a function call. Once the optimizer has processed the code, most of these uses of the alternate register set will have been removed, however some library routines do use the alternate register set.

Exit from the function is accomplished by reversing the entry code. This restores the old ix and iy, and resets the stack pointer to point to the return address, then executes a ret instruction. Example code follows:

```
pop     iy
ld      sp,ix
pop     ix
ret
```

The code for an interrupt function will also save any other registers used by the function.

2.3. Stack and Heap Allocation

As previously mentioned, the stack grows downwards. On startup, the stack pointer is initialized to the top of available memory. For CP/M this is the base of the BDOS. The heap, i.e. the area of memory dynamically allocated via sbrk, calloc or malloc, grows upwards from the top of statically allocated memory. This is defined by the top of the bss segment, which the linker gives the name __Hbss. *Sbrk()* checks the amount of memory left between the top of the heap and the bottom of the stack, and if less than 1k bytes would be left after granting the sbrk request, it will deny it. This value may be modified by editing sbrk.as, re-assembling it and replacing it in libc.lib.

2.4. Linkage to Assembler

It is quite feasible to write assembler routines to interface with C code. The following points should be noted:

1. When a routine is called from C, the arguments are pushed onto the stack in reverse order, so that the lexically first argument to the call will have the lowest memory address. The same convention must be followed by an assembler routine calling a C function.
2. When a function is called from C code, the index registers ix and iy must be preserved. If they are to be used, they should be saved on the stack and restored before returning. All other registers may be destroyed. Similarly, when

calling a C function, an assembler routine should expect only the index registers to be unmodified.

3. All C external names have an underscore prepended. Any assembler symbol to be referenced from C must start with an underscore, which should be omitted when using the name in C code.

4. Return values are in HL for words, and HL and DE for long words, with the high word in HL. Byte returns are in L, but sign or zero extended as appropriate into H.

2.4.1. Assembler Interface Example

The following function is an example of assembler code to be called from C. The function takes two integer arguments and returns the larger of the two. Note the use of the call to rcsv; this saves ix and iy and sets up ix to point to the local stack frame, then loads the first two words of the arguments into HL and DE respectively.

```
;       max(a, b)
;       returns the larger of a and b

        psect   text
        global  _max, rscv
_max:
        call    rcsv      ;a in HL, b in DE
        push    hl        ;save
        or      a         ;clear carry
        sbc     hl,de     ;compare
        pop     hl        ;restore value
        jr      nc,1f     ;skip if a greater
        ex      de,hl     ;swap
1:
        jmp     cret      ;finished
```

2.5. Data Representation

2.5.1. Longs

Long values are represented in 32 bits, with the low order word first. Within each word, the Z80 imposes a byte ordering of low byte first. This means that a long value is laid out thus:

```
-------------------------------------
| byte 0 | byte 1 | byte 2 | byte 3 |
-------------------------------------
```

where the bytes are in ascending address order, left to right.

2.5.2. Floats

Floats are also stored in 32 bits, with the least significant byte first. The high order byte consists of a sign bit (bit 7) and a 6 bit excess 64 exponent. An exponent value of 0 is stored as 64. The remaining 3 bytes are a 24-bit mantissa, in twos complement form. The floating point number is always normalized. Doubles are exactly the same as floats.

2.6. Linking Z80 Programs

The C command will normally invoke the linker to produce CP/M .COM files when compiling. If you should want to use the linker directly the following information will be useful.

2.6.0.1. Options

The psects used by the C compiler are **text**, containing program code, **data**, containing initialized data, and **bss**, containing uninitialized data. The C command supplies the linker with a -P option to concatenate these three psects into the .COM file in the order given. This option is as follows:

```
    -Ptext=0,data,bss
```

Note that the **text** psect is located at zero, while CP/M expects a .COM file to start at 100h. This is allowed for by the run-time startoff module leaving the first 100h bytes free (by a defs 100h). The **data** psect is concatenated with **text**, so the linker will start it immediately after the end of the text psect. The **bss** psect is concatenated with **data**, placing it at the end of the program. Note that since the **bss** psect should contain no initialized data, it will occupy no space in the .COM file.

The other option needed to produce a .COM file is -C. While -P tells the linker what addresses the psects should be linked for, the -C option is to tell the linker what output format is required. If the -C option is not given then the linker will create another object file containing the linked data from all the input object files. With the -C option it will produce a binary image file of the linked program. Since a .COM file is a binary image file to be loaded at 100h it is necessary to specify to the linker that the beginning of the file should represent 100h in the linked program. Thus the option required is -C100h. If for any reason there is code at an address less than 100h in the program the linker will issue an error message, since it will be attempting to write code before the beginning of the file.

2.6.0.2. Object Files

The first object file linked in a C program should always be the run-time startoff module. For CP/M this is the supplied CRTCPM.OBJ. This module contains code to set the stack pointer, clear **bss** psect, handle command line arguments and call main(). Since a CP/M program always starts execution at 100h it is important that this module comes first. The remaining object files linked are the object files that make up the program. Following these should be any libraries required, e.g. the standard library LIBC.LIB.

2.6.1. Interrupt functions

The compiler incorporates features allowing full interrupt handling without any assembler code. The type qualifier *interrupt* may be applied to a function to allow it to be called directly from a hardware (or software) interrupt. This involves code to save and restore any necessary registers, and performing an return from interrupt rather than a return from subroutine at the end of the function.

An interrupt function must be declared void and may not have parameters. It may not be called from C code, but it may call any function itself. An example of an interrupt to service a real time interrupt follows.

```
long  count;

void interrupt
real_time(void)
{
        count++;
}
```

To set interrupt vectors the routine *set_vector()* is provided. The arguments to *set_vector* are the address of the interrupt vector, casted to a pointer to a pointer to an interrupt routine, and the address of the interrupt function. This routine is declared in the header file <intrpt.h>, along with macros to enable and disable interrupts. These are *ei()* and *di()* respectively. An example follows of setting a vector for the non-maskable interrupt on the Z80

```
#include        <intrpt.h>
                /* non maskable int */
#define         NMIV     ((isr *)0x66)

static void interrupt
nmi_serv(void)
{
        /* service the interrupt */
}

static void
init_nmi(void)
{
        di();
        set_vector(NMIV, nmi_serv);
        ei();
}
```

2.6.2. Port Data Types

The compiler now allows direct access to I/O ports. It is not possible to declare variables of port type, but you can declare pointers (including constant pointers) to ports. For example:

```
port unsigned char *  pptr;
```

This variable pptr is a pointer to an unsigned char in the port data space. If the address of a port is an absolute value, you can use a typecast absolute address, e.g.

```
#define IO_PORT (*(port char *)0xE0)

func()
{
        IO_PORT = 0x40;
}
```

defines a port at the absolute address of 0xE0. This can then be used just like a variable as shown above.

2.6.3. ROM-based Programs

To produce a program to go into ROM the following steps are required: replace the standard run-time startup with one customized for your run-time environment; modify any I/O or other library routines necessary, link the run-time startoff with your program object modules then run OBJTOHEX to produce Intel hex format for PROM programming.

2.6.3.1. ROM Startup Routines

The run-time startup for a ROM based program may be based on the standard CP/M startup routine. It should set the stack pointer, clear the **bss** psect and then call the function main(). The source code for the standard startup routine is in ZCRTCPM.AS in CPM.HUF. This may be modified to produce the new run-time startup, e.g. CRTROM.AS.

2.6.3.2. Example ROM Startoff Code

An example of a run-time startoff module for a ROM system is shown below.

2.6.3.3. ROM I/O routines

If it is desired to use STDIO routines in the ROM program, e.g. printf(), the easiest place to modify the library routines is in the functions *fputc()* and *fgetc()*. These are responsible for outputting and inputting all characters that pass through the STDIO routines. The source for these is in the file STDIO.HUF.

If STDIO routines are not required then any necessary I/O routines should be written for the specific environment. The

```
;Run-time startoff for ROM

    psect text,pure
    psect data
    psect bss

RAMSIZ      equ   2048;size of  RAM
    global      _main, start, __Hbss, __Lbss

    psect text
start:
    ld     sp,__Lbss+RAMSIZ ;set up sp
    ld     de,__Lbss        ;Start of BSS
    or     a                ;clear carry
    ld     hl,__Hbss
    sbc    hl,de            ;got size
    ld     c,l
    ld     b,h
    dec    bc
    ld     l,e
    ld     h,d
    inc    de
    ld     (hl),0
    ldir         ;clear memory
    call   _main            ;call main()
    jp     start            ;restart
```

Sample ROM Startoff Code

source code for the supplied I/O routines may be useful as a starting point, e.g. the *getch()*, *putch()* and other console I/O routines.

2.6.3.4. Sample Application Code

The code below is an example of a ROM application - in this case a printer buffer which reads a serial port, buffers data and writes it out to another port.

To build this into a program the above code should be compiled to an object file, e.g. PRBUF.OBJ. Then the linker should be invoked to link the ROM startup code, the application code and the standard library (note that in this case no special I/O routines are required since all I/O is done directly to ports). The appropriate linker command line is as follows:

```
LINK -Ptext=0,data,bss=2000h CRTROM.OBJ
        PRBUF.OBJ LIBC.LIB
```

Note the following points:

1) The **bss** psect has been linked for address 2000h, which is presumably where the RAM in the system will be located. The **text** and **data** psects are concatenated into ROM starting at 0.

2) The case of the psect names is significant; if using the compiler under CP/M it will be necessary to type LINK without arguments then in response to the **link>** prompt type the rest of the command line. If the arguments are typed on the CP/M command line then CP/M will convert it to upper case. In this case the linker will complain e.g. about "unknown psect TEXT".

3) The order of the object files and library is important; the startup code must be the first object file linked, since execution will commence at the beginning of ROM.

Since no -C option was given, the output of the linker will be a file L.OBJ which contains the absolute linked program. This may then be converted to an Intel hex file as follows:

```
OBJTOHEX L.OBJ PRBUF.HEX
```

This will leave the file PRBUF.HEX containing the hex code, which may then be sent to a PROM programmer. Other output formats are available from OBJTOHEX; see the relevant manual section for more details.

```
/*
 *    Example ROM application
 *      - a printer buffer
 */

#define     IPORT 0x10   /* input data port */
#define     OPORT 0x11   /* output data port */
#define     CPORT 0x12   /* status port */
#define     IRDY  0x01   /* input ready bit */
#define     ORDY  0x02   /* output ready bit */
#define     BSIZE 16000 /* buffer size */

static charbuf[BSIZE]; /* the buffer */
unsigned    iptr, optr;

main()          /* note no argc, argv */
{
    for(;;) {
      if((iptr == BSIZE-1 ? 0 :
                 iptr+1 != optr)
        && inp(CPORT) & IRDY)
          putbuf(inp(IPORT));
      if(iptr != optr && inp(CPORT) & ORDY)
          outp(OPORT, getbuf());
    }
}

putbuf(c)
char c;
{
    if(++iptr == BSIZE)
      iptr = 0;
    buf[iptr] = c;
}

getbuf()
{
    char   c;

    c = buf[optr++];
    if(optr == BSIZE)
      optr = 0;


    }
```

3. Z80 Assembler Reference Manual

3.1. Introduction

The assembler incorporated in the HI-TECH C compiler system is a full-featured relocating macro assembler accepting Zilog mnemonics. These mnemonics and the syntax of the Z80 assembly language are described in the "Z80 Assembly Language Handbook" published by Zilog and will not be repeated here. The assembler implements certain extensions to the operands allowed, and certain additional pseudo-ops, which are described here. The assembler also accepts the additional opcodes for the Hitachi 64180 processor.

3.2. Usage

The assembler is named zas, and is invoked as follows:

```
ZAS options files ...
```

The files are one or more assembler source files which will be assembled, but note that all the files are assembled as one, not as separate files. To assemble separate files, the assembler must be invoked on each file separately. The options are zero or more options from the following list:

-N Ignore arithmetic overflow in expressions

-J Attempt to optimize jumps to branches

-U Treat undefined symbols as external

-O*file*
Place the object code in *file*

-L*list*
Place an assembly listing in the file *list*, or on standard output if *list* is null

-W*width*
The listing is to be formatted for a printer of given *width*

-C This options requests ZAS to produce cross reference information in a file. The file will be called *xxx*.crf where *xxx* is the base part of the first source file name. It will then be necessary to run the CREF utility to turn this information into a formatted listing.

Taking each of these in turn, the -N option suppresses the normal check for arithmetic overflow. The assembler follows the "Z80 Assembly Language Handbook" in its treatment of overflow, and in certain instances this can lead to an error where in fact the expression does evaluate to what the user intended. This option may be used to override the overflow checking.

The -J option will request the assembler to attempt to assemble jumps and conditional jumps as relative branches where possible. Only those conditional jumps with branch equivalents will be optimized, and jumps will only be optimized to branches where the target is in branch range. Note that the use of this option slows the assembly down, due to the necessity for the assembler to make an additional pass over the input code.

The -U option will suppress error messages relating to undefined symbols. Such symbols are treated as externals in any case. The use of this option will not alter the object code generated, but merely serves to suppress the error messages.

The default object file name is constructed from the name of the first source file. Any suffix or file type (i.e. anything following the rightmost dot ('.') in the name is stripped, and the suffix .obj appended. Thus the command

```
ZAS file1.as file2.z80
```

will produce an object file called file1.obj. The use of the -O option will override this default convention, allowing the object file to be arbitrarily named. For example:

```
ZAS -ox.obj file1.obj
```

will place the object code in x.obj.

A listfile may be produced with the -L option. If a file name is supplied to the option, the list file will be created with that name, otherwise the listing will be written to standard output (i.e. the console). List file names such as CON: and LST: are acceptable. The -W option specifies the width to which the listing is to be formatted. E.g.

```
ZAS -Llst: -W80 x.as
```

will output a listing formatted for an 80 column printer to the list device.

3.3. The Assembly Language

As mentioned above, the assembly language accepted by zas is based on the Zilog mnemonics. You should have some reference book such as the "Z80 Assembly Language Handbook". Described below are those areas in which zas differs, or has extensions, compared to the standard Zilog assembly language.

3.3.1. Symbols

The symbols (labels) accepted by the assembler may be of any length, and all characters are significant. The characters used to form a symbol may be chosen from the upper and lower case alphabetics, the digits 0-9, and the special symbols underscore ('_'), dollar ('$') and question mark ('?'). The first character may not be numeric. Upper and lower case are distinct. The following are all legal and distinct symbols.

```
An_identifier
an_identifier
an_identifier1
$$$
?$_123455
```

Note that the symbol $ is special (representing the current location) and may not be used as a label. Nor may any opcode or pseudo-op mnemonic, register name or condition code name. You should note the additional condition code names described later.

3.3.1.1. Temporary Labels

The assembler implements a system of temporary labels, useful for use within a localized section of code. These help eliminate the need to generate names for labels which are referenced only in the immediate vicinity of their definition, for example where a loop is implemented.

A temporary label takes the form of a digit string. A reference to such a label requires the same digit string, plus an appended b or f to signify a backward or forward reference respectively. Here is an example of the use of such labels.

```
entry_point:    ;This is referenced from far away
     ld   b,10
1:   dec  c
     jr   nz,2f ;if zero, branch forward to 2:
     ld   c,8
     djnz 1b    ;decrement and branch back to 1:
     jr   1f    ;this does not branch to the
                ;same label as the djnz
2:   call fred  ;get here from the jr nz,2f
1:   ret        ;get here from the jr 1f
```

The digit string may be any positive decimal number 0 to 65535. A temporary label value may be re-used any number of times. Where a reference to e.g. 1b is made, this will reference the closest label 1: found by looking backwards from the current point in the file. Similarly 23f will reference the first label 23: found by looking forwards from the current point in the file.

3.3.2. Constants

Constants may be entered in one of the radices 2, 8, 10 or 16. The default is 10. Constants in the other radices may be denoted by a trailing character drawn from the following set:

Character	Radix	Name
B	2	binary
O	8	octal
Q	8	octal
o	8	octal
q	8	octal
H	16	hexadecimal
h	16	hexadecimal

Note that a lower case b may not be used to indicate a binary number, since 1b is a backward reference to a temporary label 1:.

3.3.2.1. Character Constants

A character constant is a single character enclosed in single quotes ('). Multi character constants may be used only as an operand to a DEFM pseudo-op.

3.3.2.2. Floating Constants

A floating constant in the usual notation (e.g. 1.234 or 1234e-3) may be used as the operand to a DEFF pseudo-op.

3.3.2.3. Opcode Constants

Any z80 opcode may be used as a constant in an expression. The value of the opcode in this context will be the byte that the opcode would have assembled to if used in the normal way. If the opcode is a 2-byte opcode (CB or ED prefix byte) only the second byte of the opcode will be used. This is particularly useful when setting up jump vectors. For example:

```
ld   a,jp        ;a jump instruction
ld   (0),a       ;0 is jump to warm boot
ld   hl,boot     ;done here
ld   (1),hl
```

3.3.3. Expressions

Expressions are constructed largely as described in the "Z80 Assembly Language Handbook".

3.3.3.1. Operators

The following operators may be used in expressions:

Operator	Meaning
&	Bitwise AND
*	Multiplication
+	Addition
-	Subtraction
.and.	Bitwise AND
.eq.	Equality test
.gt.	Signed greater than
.high.	Hi byte of operand
.low.	Low byte of operand
.lt.	Signed less than
.mod.	Modulus
.not.	Bitwise complement
.or.	Bitwise or
.shl.	Shift left

.shr.	Shift right
.ult.	Unsigned less than
.ugt.	Unsigned greater than
.xor.	Exclusive or
/	Divison
<	Signed less than
=	Equality
>	Signed greater than
^	Bitwise or

3.3.3.2. Relocatability

Zas produces object code which is relocatable; this means that it is not necessary to specify assembly time where the code is to be located in memory. It is possible to do so, by use of the ORG pseudo-op, however the preferred approach is to use program sections or psects. A psect is a named section of the program, in which code or data may be defined at assembly time. All parts of a psect will be loaded contiguously into memory, even if they were defined in separate files, or in the same file but separated by code for another psect. For example, the following code will load some executable instructions into the psect named text, and some data bytes into the data psect.

```
        psect   text, global

alabel:
        ld      hl,astring
        call    putit
        ld      hl,anotherstring

        psect   data, global
astring:
        defm    'A string of chars'
        defb    0
anotherstring:
        defm    'Another string'
        defb    0

        psect   text

putit:
        ld      a,(hl)
        or      a
        ret     z
        call    outchar
        inc     hl
        jr      putit
```

Note that even though the two blocks of code in the text psect are separated by a block in the data psect, the two text psect blocks will be contiguous when loaded by the linker. The instruction "ld hl,anotherstring" will fall through to the label "putit:" during execution. The actual location in memory of the two psects will be determined by the linker. See the linker manual for information on how psect addresses are determined.

A label defined in a psect is said to be relocatable, i.e. its actual memory address is not determined at assembly time. Note that this does not apply if the label is in the default (unnamed) psect, or in a psect declared absolute (see the PSECT pseudo-op description below). Any labels declared in an absolute psect will be absolute, i.e. their address will be determined by the assembler.

Relocatable symbols may be combined in expressions, but only within certain constraints. Briefly, the following combinations are legal, where A is an absolute expression, R1 and R2 are relocatable expressions:

R1 + A	is relocatable
A + R1	is relocatable
R1 - A	is relocatable
R1 - R2	is absolute

Note that the expression R1 - R2 is legal if and only if R1 and R2 are defined in the same psect. All other combinations of relocatable quantities are illegal.

3.3.4. Pseudo-ops

The pseudo-ops are based on those described in the "Z80 Assembly Language Handbook", with some additions.

3.3.4.1. DEFB

This pseudo-op should be followed by a comma-separated list of expressions, which will be assembled into sequential byte locations. Each expression must have a value between -128 and 255 inclusive. Example:

```
DEFB    10, 20, 'a', 0FFH
```

3.3.4.2. DEFW

This operates in a similar fashion to DEFB, except that it assembles expressions into words, without the value restriction. Example:

```
DEFW    -1, 3664H, 'A', 3777Q
```

3.3.4.3. DEFS

Defs reserves memory locations without initializing them. Its operand is an absolute expression, representing the number of bytes to be reserved. This expression is added to the current location counter. Note however that locations reserved by DEFS may be initialized to zero by the linker if the reserved locations are in the middle of the program. Example:

```
DEFS    20h      ; reserve 16 bytes of memory
```

3.3.4.4. EQU

Equ sets the value of a symbol on the left of EQU to the expression on the right. It is illegal to set the value of a symbol which is already defined. Example:

```
SIZE  equ   46
```

3.3.4.5. DEFL

This is identical to EQU except that it may redefine existing symbols. Example:

```
SIZE  defl  48
```

3.3.4.6. DEFM

Defm should be followed by a string of characters, enclosed in single quotes. The ASCII values of these characters are assembled into successive memory locations. Example:

```
DEFM    'A string of funny *@$ characters'
```

3.3.4.7. END

The end of an assembly is signified by the end of the source file, or the END pseudo-op. The END pseudo-op may optionally be followed by an expression which will define the start address of the program. This is not actually useful for CP/M. Only one start address may be defined per program, and the linker will complain if there are more. Example:

```
END    somelabel
```

3.3.4.8. COND

Conditional assembly is introduced by the COND pseudo-op. The operand to COND must be an absolute expression. If its value is false (i.e. zero) the code following the COND up to the corresponding ENDC pseudo-op will not be assembled. COND/ENDC pairs may be nested. Example:

```
COND   Debug
call   trace    ;trace execution if debugging
ENDC
```

3.3.4.9. ENDC

See COND.

3.3.4.10. ENDM

See MACRO.

3.3.4.11. PSECT

This pseudo-op allows specification of relocatable program sections. Its arguments are a psect name, optionally followed by a list of psect flags. The psect name is a symbol constructed according to the same rules as for labels, however a psect may have the same name as a label without conflict. Psect names are recognized only after a PSECT pseudo-op. The psect flags are as follows:

ABS	Psect is absolute
GLOBAL	Psect is global
LOCAL	Psect is not global
OVRLD	Psect is to be overlapped by linker
PURE	Psect is to be read-only

If a psect is global, the linker will merge it with any other global psects of the same name from other modules. Local psects will be treated as distinct from any other psect from another module. Psects are global by default.

By default the linker concatenates code within a psect from various modules. If a psect is specified as OVRLD, the linker will overlap each module's contribution to that psect. This is particularly useful when linking modules which initialize e.g. interrupt vectors.

The PURE flag instructs the linker that the psect is to be made read-only at run time. The usefulness of this flag depends on the ability of the linker to enforce the requirement. CP/M fails miserably in this regard.

The ABS flag makes a psect absolute. The psect will be loaded at zero. This is useful for something, but I cannot quite remember what it is. Examples:

```
PSECT   text, global, pure
PSECT   data, global
PSECT   vectors, ovrld
```

3.3.4.12. GLOBAL

Global should be followed by one more symbols (comma separated) which will be treated by the assembler as global symbols, either internal or external depending on whether they are defined within the current module or not. Example:

```
GLOBAL label1, putchar, _printf
```

3.3.4.13. ORG

An ORG pseudo-op sets the current psect to the default (absolute) psect, and the location counter to its operand, which must be an absolute expression. Example:

```
ORG    100H
```

3.3.4.14. MACRO

This pseudo-op defines a macro. It should be followed by the macro name, then optionally by a comma-separated list of formal parameters. The lines of code following the MACRO pseudo-op up till the next ENDM pseudo-op will be stored as the body of the macro. The macro name may subsequently be used in the opcode part of an assembler statement, followed by actual parameters. The text of the body of the macro will be substituted at that point, with any use of the formal parameters substituted with the corresponding actual parameter. For example:

```
MACRO  bdos, func, arg
ld     de,arg
ld     c,func
call   5
ENDM
```

When used, this macro will expand to the 3 instructions in the body of the macro, with the actual parameters substituted for func and arg. Thus

```
        bdos    1, (charbuf)

    expands to

        ld      de,(charbuf)
        ld      c,1
        call    5
```

3.3.5. Extended Condition Codes

The assembler recognizes several additional condition codes. These are:

Code	Equivalent	Meaning
alt	m	Arithmetic less than
llt	c	Logical less than
age	p	Arithmetic greater or equal
lge	nc	Logical greater or equal
di ei		Use after ld a,i for testing state of interrupt enable flag - enabled or disabled respectively.

3.4. Assembler Directives

An assembler directive is a line in the source file which produces no code, but rather which modifies the behaviour of the assembler. Each directive is recognized by the presence of an asterisk in the first column of the line, followed immediately by a word, only the first character of which is looked at. the line containing the directive itself is never listed. The directives are:

*Title

Use the text following the directive as a title for the listing.

*Heading

Use the text following the directive as a subtitle for the listing; also causes an *Eject.

*List

May be followed by ON or OFF to turn listing on or off respectively. Note that this directive may be used inside a macro or include file to control listing of that macro or include file. The previous listing state will be restored on exit from the macro or include file.

*Include

The file named following the directive will be included in the assembly at that point.

*Eject

A new page will be started in the listing at that point. A form feed character in the source will have the same effect.

Some examples of the use of these directives:

```
*Title Widget Control Program
*Heading Initialization Phase

*Include widget.i
```

3.5. Diagnostics

An error message will be written on the standard error stream for each error encountered in the assembly. This message identifies the file name and line number and describes the error. In addition the line in the listing where the error occurred will be flagged with a single character to indicate the error. The characters and the corresponding messages are:

A:	**Absolute expression required**
B:	**Bad arg to *L**
	Bad arg to IM
	Bad bit number
	Bad character constant
	Bad jump condition
D:	**Directive not recognized**
	Digit out of range
E:	**EOF inside conditional**
	Expression error
G:	**Garbage after operands**
	Garbage on end of line
I:	**Index offset too large**
J:	**Jump target out of range**
L:	**Lexical error**
M:	**Multiply defined symbol**
O:	**Operand error**
P:	**Phase error**
	Psect may not be local and global
R:	**Relocation error**
S:	**Size error**
	Syntax error
U:	**Undefined symbol**
	Undefined temporary label
	Unterminated string

4. Lucifer - A Source Level Debugger

4.1. Introduction

Lucifer is a source level remote debugger for use with the HI-TECH C compilers. It consists of a program which runs on a host machine (usually MS-DOS or Unix) which communicates with a target system via a serial line or an in-circuit emulator.

The host program provides the user interface, including source code display, disassembly, displaying memory etc. The target system must have logic to read and write memory and registers, and implement single stepping. With each version of Lucifer a small program is provided which can be compiled and placed in a ROM in a target system to implement these features. The standard host program is set up to communicate with this ROM program via a serial line.

Alternatively the host program can be modified to communicate with other systems, e.g. in-circuit emulators.

4.2. Usage

To use Lucifer you should compile your program with a -G option. This will produce a symbol file with line number and filename symbols included. If you use a -H option you will get a symbol file but without the filename and line number symbols.

Example, to compile "test.c" and produce a symbol file "test.sym":

```
ZC -GTEST.SYM TEST.C
```

Then invoke Lucifer as follows:

```
LUCIFER COMn L.SYM
```

COMn should be COM1 or COM2 if you are using an IBM PC or clone. Lucifer will access an IBM standard serial port addressed as either port. For Unix simply specify the name of the serial port connected to your target, e.g. */dev/tty006*. The default baud rate is 19200. You can override this with a -S option, e.g.

```
LUCIFER -S9600 COM1 L.SYM
```

will access COM1 at 9600 baud.

Lucifer should announce itself, then attempt to communicate with the target. If successful, it prints a message sent by the target identifying itself. If Lucifer has started succesfully you should see a display something like this:

```
B>lucifer com1 test.sym
LUCIFER SOURCE LEVEL DEBUGGER (Z80) V3.00
Copyright (C) 1989 HI-TECH SOFTWARE

Machine = Z80, target identifies as
Z80 Lucifer v3.00
:
```

It then prints a colon (':') and waits for commands. For a list of commands, type the command **h**. Note that all commands should be in lower case. Symbols should be entered in exactly the same case as they were defined. Where an expression is required, it may be of the forms:

symbol_name
symbol+hexnum
$hexnum
:linenumber

i.e. a symbol name (e.g. *main*), a symbol name plus a hex offset (e.g. *afunc+1a*), a hex number preceded by a dollar sign, or a line number preceded by a colon. In the **b** (breakpoint) command any decimal number will be interpreted as a line number by default, while in the **u** (unassemble) command any number will be intepreted as a hex number representing an address by default. These assumptions can always be overridden by using the colon or dollar prefixes.

When entering a symbol, it is not necessary to type the underscore prepended by the C compiler, however when printing out symbols the debugger will always print the underscore. Any register name may also be used where a symbol is expected.

4.3. Commands

4.3.1. The B Command: set or display breakpoints

The **b** command is used to set and display breakpoints. If no expression is supplied after the **b** command, a list of all currently set breakpoints will be displayed. If an expression is supplied, a breakpoint will be set at the line or address specified. If you attempt to set a breakpoint which already exists, or enter an expression which Lucifer cannot understand, an appropriate error message will be displayed. Note: any decimal number specified will be interpreted as a line number by default, if you want to specify an absolute address, prefix it with a dollar sign.

```
: b 10
Set breakpoint at _main+$27
:

Example: (setting a breakpoint at line
             10 of a C program)
```

4.3.2. The D Command: display memory contents

The **d** command is used to display a hex dump of the contents of memory on the target system. If no expressions are specified, one byte is dumped from the address reached by the last **d** command. If one address is specified, 16 bytes are dumped from the address given. If two addresses are specified, the contents of memory from the first address to the second address are displayed. Dump addresses given can be symbols, line numbers, register names or absolute memory addresses.

4.3.3. The E Command: examine C source code

The **e** command is used to examine the C source code of a function or file. If a function name is given, Lucifer will locate the source file containing the function requested and display from just above the start of the function. If a file name is given, Lucifer will display from line 1 of the requested file.

```
: e main
2:
3:int value, result;
4:
5:main()
6:{
7:     scanf("%d",&value);
8:     result = (value << 1) + 6;
9:     printf("result = %d\n",result);
10:}
:
```

Example: (use of e command to list C *main()* function)

4.3.4. The G Command: commence execution

The g command is used to commence execution of code on the target system. If no expression is supplied after the g command, execution will commence from the current value of PC (the program counter). If an expression is supplied, execution will commence from the address given. Execution will continue until a breakpoint is reached, or the user interrupts with control-C. After a breakpoint has been reached, execution can be continued from the same place using the g, s and t commands.

4.3.5. The I Command: toggle instruction trace mode

The i command is used to toggle instruction trace mode. If instruction trace mode is enabled, each instruction is displayed before it is executed while stepping by entire C lines with the s command.

```
Assume PC points to 9:
printf("result = %d\n",result);

With instruction trace turned OFF,
stepping would produce:

: s
result = 20
Stepped to
10:}
:

With instruction trace turned ON,
stepping would produce:

: s
_main+$1B      LD       HL,(_result)
_main+$1E      PUSH     HL
_main+$1F      LD       HL,__Ldata+$5
_main+$22      PUSH     HL
_main+$23      CALL     _printf
result = 20
_main+$26      POP      BC
_main+$27      POP      BC
Stepped to
10:}
:
```

Note: the library function *printf()* was not traced by the stepping mechanism and thus functioned correctly.

Example: (showing effect of i command on step display)

4.3.6. The L Command: load a hex file

The l command is used to load object files into the target system. Lucifer expects either Intel Hex or Motorola S-record format object files and will refuse to load files which are not in one of those formats.

4.3.7. The P Command: toggle input prompting mode

The **p** command is used to toggle input prompting. If input prompting is enabled, Lucifer will display a prompt "Target wants input:" when the target program executes an input function (*gets(), scanf(), etc.*). If input prompting is disabled, input prompting is left to the target program.

4.3.8. The Q Command: exit to operating system

The **q** command is used to exit from Lucifer to the operating system. Note: the **q** command does **not** stop the target program (i.e. the Lucifer monitor running on the target system), so it is possible to re-enter Lucifer without re-initialising the target.

4.3.9. The R Command: remove breakpoints

The **r** command is used to remove breakpoints which have been set with the **b** command. For each breakpoint, the user is prompted as to whether the breakpoint should be removed or not.

```
: r
Remove _main+$27 ? y
Remove _main+$44 ? n
Remove _test ? n
:

Example: (removes breakpoint at main+$27)
```

4.3.10. The S Command: step one C line

The **s** command is used to step by one line of C source code. This is normally implemented by executing several machine instruction single steps, and therefore can be quite slow. If Lucifer can determine that there are no function calls or control structures (*break, continue, etc.*) in the line, it will set a temporary breakpoint on the next line and execute the line at full speed.

When single stepping by machine instructions, the step command will execute subroutine calls to external and library functions at full speed, thereby avoiding the slow process of single stepping through complex library routines such as *printf()* or *scanf()*. Normal library console I/O works correctly during single stepping.

```
Assume current PC points to line 6:
{

: s
Stepped to
7:    scanf("%d",&value);
: s
Target wants input: 7
Stepped to
8:    result = (value << 1) + 6;
: s
Stepped to
9:    printf("result = %d\n",result);
: s
result = 20
Stepped to
10:}
:
```

Example: (stepping through several lines of C code)

4.3.11. The T Command: trace one instruction

The **t** command is used to trace one machine instruction on the target. The current value of PC (the program counter) is used as the address of the instruction to be executed. After the instruction has been executed, the next instruction and the contents of all registers will be displayed.

4.3.12. The U Command: disassemble

The **u** command disassembles object code from the target system's memory. If an expression is supplied, the disassembly commences from the address supplied. If an address is not supplied, the disassembly commences from the instruction where the last disassembly ended. The disassembler automatically converts addresses in the object code to symbols if the symbol table for the program being disassembled is available. If the source code for a C program being disassembled is available, the C lines corresponding to each group of instructions are also displayed. Note: any values specified will be interpreted as absolute addresses by default, if you want to specify a line number, prefix it with a colon.

```
: u :8
8:    result = (value << 1) + 6;
_main+$10      LD         DE,0006H
_main+$13      LD         HL,(_value)
_main+$16      ADD        HL,HL
_main+$17      ADD        HL,DE
_main+$18      LD         (_result),HL
9:    printf("result = %d\n",result);
_main+$1B      LD         HL,(_result)
_main+$1E      PUSH       HL
_main+$1F      LD         HL,_Ldata+$6
_main+$22      PUSH       HL
_main+$23      CALL       _printf
_main+$26      POP        BC
_main+$27      POP        BC
:
```

Example: (disassembly from line 8 of a C program)

4.3.13. The X Command: examine or change registers

The x command is used to examine and change the contents of the target CPU registers. If no parameters are given, the registers are displayed without change. To change the contents of a register, two parameters must be supplied, a valid register name and the new value of the register. After setting a new register value, the contents of the registers are displayed.

```
:x ix 1234 hl 3fff
```

Example: (use of x command to set the IX register to 1234H and the BC register to 3fffH)

Any valid Z80 register name may be used with the x command. The alternate register set may be accessed by appending a single quote ' onto the name of the register, for example:

`bc'`	is the alternate BC register
l'	is the alternate L register

4.3.14. The . Command: set a breakpoint and go

The . command is used to set a temporary breakpoint and resume execution from the current value of PC (the program counter). Execution continues until any breakpoint is reached, or the user interrupts with control-C, then the temporary breakpoint is removed. Note: the temporary breakpoint is removed even if execution stops at a different breakpoint or is interrupted. If no breakpoint address is specified, the . command will display a list of active breakpoints.

```
Assume PC points to the start of the
program.

: . 10
Target wants input: 7
result = 20

Breakpoint
10:}
_main+$28     RET
PC = 01F2  .....  AF = 0044 -Z---P--
IX = 6F1C  .....  AF'= 2809 -------C
:

Example: (use of . command to execute to
         line 10 of a C program)
```

4.3.15. The ; Command: display from a source line

The ; command is used to display 10 lines of source code from a specified position in a source file. If the line number is omitted, the last page of source code displayed will be redisplayed. Example:

```
: ; 4
4:
5: main()
6: {
7:      scanf("%d",&value);
8:      result = (value << 1) + 6;
9:      printf("result = %d\n",result);
10:}
```

4.3.16. The = Command: display next page of source

The = Command is used to display the next 10 lines of source code from the current file. For example, if the last source line displayed was line 7, = will display 10 lines starting from line 8.

4.3.17. The - Command: display previous page of source

The - Command is used to display the previous 10 lines of source code from the current file. For example, if the last page displayed started at line 15, - will display 10 lines starting from line 5.

4.3.18. The / Command: search source file for a string

The / command is used to search the current source file for occurences of a sequence of characters. Any text typed after the / is used to search the source file. The first source line containing the string specified is displayed. If no text is typed after the /, the previous search string will be used. Each string search starts from the point where the previous one finished, allowing the user to step through a source file finding all occurences of a string.

```
: /printf
10:     printf("Enter a number:");
: /
14:     printf("Result = %d\n",answer);
:

Example: (use of / command to find
         uses of printf() )
```

4.3.19. The ! Command: execute a DOS command

The ! command is used to execute an operating system shell command line without exiting from Lucifer. Any text typed after the ! is passed through to the shell without modification.

4.3.20. Other Commands

In addition to the commands listed above, Lucifer will interpret any valid decimal number typed as a source line number and attempt to display the C source code for that line.

Pressing return without entering a command will result in re-execution of the previous command if the previous command was **d**,**e**,**s**, **t** or **u**. In all cases the command resumes where the previous one left off. For example, if the previous command was **d 2000**, pressing return will have the same effect as the command **d 2010**.

4.4. Installing Lucifer on a Target

In order to use Lucifer on your target system, you will need to compile the Lucifer monitor TARGET.C and place it in a ROM. If your Z80 system already has a monitor in ROM, it is also possible to download the Lucifer target code into RAM. If you have one of the Z80 boards supported by the supplied target code, you will be able to use the Lucifer monitor program supplied without modification, however, if your Z80 system is not one of the ones supported, you will need to modify the target code before using it. Areas which will usually need modification are the serial port drivers and the single step code. As supplied, the single step code is configured to use the single step NMI system on the JED STD-801 board.

To compile the target program use the command:

```
ZC -O -OTARGET.HEX -Arom,ram,ramsize
TARGET.C
```

where

```
rom  =  the address in ROM where the
        monitor will reside.

ram  =  the address in RAM where the
        monitor's global variables and
        stack will be located

size =  the size of the monitors
        global variable and stack area
```

Example:

```
To compile TARGET.C to run at address
0F000H with 800H bytes of RAM at 7800H:

ZC -O -OTARGET.HEX -AF000,7800,800
TARGET.C
```

The compiler will create TARGET.HEX, an Intel format hex file, ready to be programmed into a ROM.

4.4.1. Modifying the Target Program

Most modifications to TARGET.C will be made to the serial I/O functions, *putch()*, *getch()* and *init_uart()* which are pre-configured to use the EXAR 88681 UART on the JED STD-801 board. On systems where there is no single step interrupt available, single stepping may be achieved by setting a breakpoint at the end point of each instruction before it is executed. For conditional branches, two strategies are available:

1) Pre-evaluate the condition and determine which path will be taken.

2) Set a breakpoint at BOTH possible end points of the instruction.

If you have modified TARGET.C to run on a Z80 system which we don't currently support, please send the modified code to us to assist us in building up our library of supported systems.

4.5. CP/M BDOS Call Support

The Lucifer target code supplied will emulate the CP/M BDOS calls which handle console I/O if required to. If TARGET.C is compiled with the flag **-DCPMCALLS**, the BDOS emulation code will be included. BDOS emulation is useful on systems which do not otherwise support a console device.

To compile TARGET.C with the same addresses as before, but BDOS emulation enabled:

```
ZC -O -OTARGET.HEX -AF000,7800,800
TARGET.C -DCPMCALLS
```

If TARGET.C finds a JP instruction already installed at location 5, it will re-direct the JP to its own handler, and store the old BDOS vector for use with BDOS calls which it does not support. The CP/M calls which are supported are:

```
1       CONSOLE_INPUT
2       CONSOLE_OUTPUT
6       DIRECT_CONSOLE_IO
10      READ_CONSOLE_BUFFER
11      GET_CONSOLE_STATUS
```

All other calls are passed through to the old BDOS handler. If a BDOS handler was not found at location 5, any unimplemented BDOS calls cause the target code to print an error message via the serial port. It would be feasible to modify TARGET.C to include an entire CP/M BDOS emulation including access to files on the host system via the serial link. We have not implemented remote file server support as in most cases, if file access is desired, the user will be running on a CP/M system anyway.

INDEX