```
          Small-C/Plus Compiler Documentation

INTRODUCTION

This compiler is based on the Small-C compiler written by Ron
Cain and published in Dr. Dobb's #45 (May '80).  The compiler
was modified to include floating point by James R. Van Zandt.
The floating point routines themselves were written by Neil
Colvin.  Further improvements, especially in control structures,
code optimisation, additional data types, structures and unions
were added by Ronald M. Yorston.  In part, these improvements
are based on the work of James E. Hendrix.  The companion
assembler ZMAC and linker ZLINK were written by Bruce Mallett.
The library reference resolver, ZRES, the library manager, ZLIB,
and the assembler optimiser, ZOPT, were written by Ron Yorston.

This compiler accepts a subset of standard C.  It requires a
Z-80 processor.  It reads C source code and produces Zilog
mnemonic assembly language output, with syntax matching the
assembler ZMAC supplied with it.  ZMAC produces a relocatable
file with the extension .OBJ.  One or more such relocatable
files can be linked with the companion program ZLINK.  All the
programs in the suite (CC0, ZMAC, ZLINK, ZRES, ZLIB, ZOPT) are
in the public domain.

For more about the C programming language, see  "The C
Programming Language" by B. W. Kernighan and D. M. Ritchie, 1978
(ISBN 0-13-110163-3).  For more information on the Small-C
compiler see "The Small-C Handbook" by James E. Hendrix, 1984
(ISBN 0-8359-7012-4).  Another useful book is "Dr Dobb's
Toolbook of C", 1986 (ISBN 0-89303-615-3).


DATA TYPES

The data types are...

char c;           character
char *c;          pointer to character
char **c;         pointer to pointer to character
char c();         function returning character
char *c();        function returning pointer to character
char c[3];        character array
char *c[3];       array of pointers to character

int i;            16 bit integer
int *i;           pointer to integer
int **i;          pointer to pointer to integer
int i();          function returning integer
int *i();         function returning pointer to integer
int i[4];         integer array
int *i[4];        array of pointers to integer
int (*i)();       pointer to function returning integer

double d;         48 bit floating point
double *d;        pointer to double
double **d;       pointer to pointer to double
double d();       function returning double
double *d();      function returning pointer to double
double d[5];      array of doubles
double *d[5];     array of pointers to double
```



```
struct st s;      structure with tag st
struct st *s;     pointer to structure with tag st
struct st **s;    pointer to pointer to structure
struct st *s();   function returning pointer to structure
struct st s[5];   array of structures with tag st
struct st *s[5];  array of pointers to structure

union un u;       union with tag un
union un *u;      pointer to union with tag un
union un **u;     pointer to pointer to union with tag un
union un *u();    function returning pointer to union
union un u[5];    array of union with tag un
union un *u[5];   array of pointers to union with tag un

Characters are 8-bit signed integers with values in the range -128
to 127.  Integers are signed integers with values between -32768
and 32767.  Pointers contain the addresses of data elements and
are treated as unsigned integers when compared.

Storage classes (other than extern), multidimensional arrays,
and more complex types like "int *(*i)()" (pointer to function
returning pointer to int) are not included.  Although multi-
dimensional arrays are not allowed, it is possible to use arrays
of pointers to produce a similar effect.

Structures may be declared using the syntax:

        struct st {
                int x ;
                char c[12] ;
                struct other s ;
                struct st *st_pointer ;
        } x, *y, z[5] ;

The structure tag is compulsory.  Structure declarations may be
nested, although the members of the nested structures then share
the same name space.  Structures may not contain instances of
themselves, but can contain pointers to instances of themselves.
The members of a structure must be declared the first time the
structure is mentioned.   Structure passing and assignment are
not supported.  This means that the only legal occurrence of a
structure in a function argument list is as a pointer.
Structure tags, structure members and ordinary variables have
separate name spaces.  Unions are treated as a special form of
structure, so all the same rules and restrictions apply.


PRIMARIES

        array[expression]
        function(arg1,arg2,...,argn)
        structure.member
        st_pointer->member
        constant
                decimal integer
                hexadecimal integer (0xfe, 0X12CD, 0x04A)
                decimal floating point (1.0, 2., .3, 340.2e-8)
                quoted string    ("sample string")
                primed character ('a'  or  'Z')
                sizeof()
        local variable
        global variable
```

Each variable must be declared before it is used. Variables declared outside a function are global. Local variables may be declared at the start of any compound statement, except the compound statement of a switch statement. The scope of local variables is restricted to the compound statement or function in which they are declared. The extern keyword may be used to declare external variables and functions. Only the first eight characters of variable and function names are significant.

The following global objects may be initialised when they are declared: chars, ints, pointers to char, arrays of char, arrays of int, arrays of pointer to char, structs, pointers to struct and arrays of struct. Doubles may not be initialised. Some examples of initialisation:

```
        char c = 'a' ;
        int i = 0 ;
        char *message = "Hello world\n" ;
        char bye[] = "Goodbye" ;
        int x[7] = { 0, 1, 2, 3, 4, 5, 6 } ;
        char *mon[] = { "jan", "feb", "mar", 0 } ;

        struct st { char c; int x; char *cp } ;
        struct st st1 = { 'q', 23, "a string"} ;
        struct st *st2 = { 'r', 42, "another string"} ;
        struct st st3[] = {
                        {'a', 1, "a"},
                        {'b', 2, "b"} } ;
```

Arrays being initialised may have their dimension specified, in which case any uninitialised entries are set to zero. Alternatively, the array dimension may be left blank, in which case the size of the array is determined by the number of initialisers present. Uninitialised global variables are set to zero. Local variables cannot be initialised and their initial contents are undefined.

Character pointers may only be initialised to a string constant or zero. The only structure members which can be initialised are chars, ints and pointers to char. A structure containing any other type of member cannot be initialised.

Within quotes or single inverted commas the escape sequences '\b', '\t', '\l', '\f' and '\n' may be used to represent 8, 9, 10, 12 and 13 respectively. Note that '\n' represents a carriage return, not a line feed as is more often the case. No problems will arise if '\n' is used where a newline is required. An arbitrary character can be represented by a three digit octal sequence: '\ooo'. Any other character following a backslash is removed of its special meaning, so a double quote may be included in a quoted string by saying '\"', and a backslash by saying '\\'.

The value of a quoted string is a pointer to the string, terminated by a null byte, somewhere in memory.

The sizeof() operator returns the size of an object in bytes. It may only take arguments of the form "int", "char", "double" or "struct st". Variable names are not acceptable.

UNARY INTEGER OPERATORS

!	logical not
~	ones complement
-	minus
*	indirection
&	address of
++	increment, either prefix or suffix
--	decrement, either prefix or suffix

BINARY INTEGER OPERATORS

+	addition
-	subtraction
*	multiplication
/	division
%	mod, remainder from division
|	inclusive or
^	exclusive or
&	logical and
<<	left shift
>>	arithmetic right shift
&&	logical and
||	logical or
=	assignment

UNARY DOUBLE OPERATORS

-	minus
*	indirection
&	address of

BINARY DOUBLE OPERATORS

+	add
-	subtract
*	multiply
/	divide
=	assignment

RELATIONAL OPERATORS

==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

ASSIGNMENT OPERATORS

INTEGER

+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=

DOUBLE

```
                +=, -=, *=, /=


CONDITIONAL OPERATOR

        ?:      conditional operator


Conversion between floating point and integer is automatic for
assignment and for the expression returned by a function.
Conversion from integer variables to floating point is
automatic for the arguments of any of the floating point
operators.  Otherwise, the routines "float(jj)" and "ifix(yy)"
(as in FORTRAN) may be used.  Integer constants may not be used
in floating point expressions; the compiler warns of this
situation.  The arguments of integer-only operators are checked
to ensure they are integers.  There is no type checking for the
actual parameters of function calls.

When adding an integer to a pointer, the increment is scaled by
the size of the object pointed to.  Thus, adding n to a pointer
makes it point to the nth object along in memory, regardless of
the size of the object involved.   When two pointers (to objects
of the same type) are subtracted the difference is scaled down
by the size of the object pointed to.  The result only makes
sense if the pointers refer to the same array:  it then gives
the number of elements between the pointers.

The comma operator may be used to separate expressions in an
expression list.  The expressions in the list are evaluated left
to right and the value returned is that of the rightmost
expression.


TYPES OF STATEMENT

expression;                     Expression statement

if(expression) statement;       Statement executed if expression
                                is non-zero

if(expression) statement;       Statement executed if expression
                                is non-zero
else statement;                 Statement executed if expression
                                is zero

while(expression)statement;     Statement executed while expression
                                is non-zero.  (It is possible for
                                the statement not to be executed
                                at all.)

do statement while(expression); Do statement until expression is
                                false.  The test is at the end of
                                the loop.

for(expr1;expr2;expr3)          expr1 initialises a variable.
  statement ;                   expr2 tests a condition involving
                                the variable.  expr3 increments or
                                otherwise alters the variable. The
                                statement is executed until expr2
                                becomes false.
```



```
switch(expression) {            Different case statements are
case value1:statement;          executed depending on the value of
case value2:statement;          the switch expression.  Most case
 etc.                           statements end with a break to avoid
default:statement;              the other statements below.  The
}                               switch expression must be an
                                integer and the case values must
                                be integer constant expressions.

break;                          Control transferred from the
                                innermost loop or switch

continue;                       Return control to the loop-
                                continuation portion of the
                                enclosing while, do or for loop

return;                         Return from function

return expression;              Return from function with value
                                given by expression

 ;                              Null statement

{statement;statement;           Compound statement which may be used
 statement;..statement;}        anywhere instead of a simple
                                statement.


If functions return anything other than an integer, they must be
declared before use in each compilation.  Otherwise, functions
are automatically imported and exported.  Names of functions and
global variables (i.e., those declared outside function
definitions) are always global as far as the linker is
concerned, and may not overlap.  (i.e. there are no static
functions or variables.)


COMMAND LINE ARGUMENTS

The function 'main' can start with:

        main(argc,argv)
        int argc ;
        char **argv ;
        {

Argc is an integer equal to the number of command line arguments
and will be equal to one if the command line consisted only of a
command.  Argv is an array of pointers to character strings
which are initialised with the command line arguments.  Argv[0]
would contain the command name, but CP/M does not allow access
to that information, so it points to a null string instead.
Argv[1] contains a pointer to the first argument, etc.


EMBEDDED COMPILER COMMANDS

The following pseudo-preprocessor directives are recognised:

        #define name string

'name' is replaced by 'string' hereafter.  Macro arguments are
```

```
not permitted.

        #undef name

The definition for the macro 'name' is removed from the macro
table.

        #ifdef name
        #ifndef name
        #else
        #endif

The above directives allow conditional compilation.  If 'name'
is defined, code between #ifdef name and #else or #endif is
compiled.  If 'name' is not defined the code between #ifdef name
and #else or #endif is not compiled.  The #else directive
toggles whether or not code is compiled.  The #endif directive
terminates an #ifdef or #ifndef block.  The #ifndef directive
works in the opposite sense to #ifdef, compiling code if 'name'
is not defined, and ignoring it if it is.


        #include filename

compiler gets source code from another file (can't be nested)

        #asm
        ...
        ...
        #endasm

code between these is passed directly to the assembler.  The
only preprocessor directives which apply between #asm and
#endasm are the conditional compilation directives.  This allows
conditional assembly based on the value of a #defined constant.

Comments are written: /* comment */
Comments may not be nested.


USING THE COMPILER

When the compiler is run, it reads one or more C source files
and produces one assembly language file.  Assembly language
files are separately assembled by ZMAC, references to library
routines are resolved by ZRES, and then a single executable file
is built by ZLINK.

The format of the compiler command line is:

        cc0 [options] file [file file...]

Each option is a minus sign followed by a letter:

        -C      include the C source code as comments in the
                compiler-generated assembly code.

        -Dname[=value]
                define the symbolic value 'name'.

        -E      pause after an error is encountered.
```



```
        -M      none of the named files contains main().

        -T      enable walkback trace on calls to err().

        -Uname  undefine the macro 'name'.

The -D options makes it possible to define symbolic values at
compile time.  For example, you could define the symbol DEBUG to
include debugging code in the compiled program, using the
conditional compilation features of the preprocessor.  If the
'value' is omitted the symbol takes the value 1.  Note that
because CP/M translates the command line into upper case it is
only possible to define upper case symbols and values.  The
symbols CPM, Z80, PCW and SMALL_C are predefined.

The -M option stops the compiler from producing its standard
header (initializing the stack pointer, for example), which is
only required in the first object module to be linked.  The
header does not include an ORG 100H directive, since ZLINK
automatically starts programs at 100H.  As a result, forgetting
the -M option will lengthen your program by a few bytes but
cause no other harm.

The -T option compiles code into each function which will allow
a "walkback trace" to be printed when err() is called.  The
walkback trace lists all the functions that have been called but
which have not yet returned (recursive calls lead to multiple
listings).

The -U option removes a macro definition from the macro table.
It can be used to undefine the predefined symbols CPM, Z80, PCW
and SMALL_C.

Options and files are separated by spaces, and options must
precede file names.  Only file names (optionally preceded by a
disk name) should be given:  the compiler automatically adds the
extension ".C".  The output file is given the same name (and is
put onto the same disk) as the first input file, but with the
extension ".ASM".

Each assembly language file is assembled as follows:

        zmac alpha=alpha

If extensions are not specified, as here, ZMAC uses ".ASM" for
its input file and ".OBJ" for its output file.

References in the ".OBJ" files to routines in the library are
resolved using ZRES:

        zres b: clib alpha beta

This command will scan the library clib to resolve references
contained in the files ALPHA.OBJ and BETA.OBJ.  A submit file
will be generated to perform the necessary linkage.  The
standard run-time routines in IOLIB.OBJ is always included, and
any other library routines are copied into the temporary file
CLIB.OBJ.  The submit file also changes to drive B: and copies
the resulting executable there.

The object files are linked as follows:
```

```
        zlink alpha,alpha=alpha,beta,iolib,clib

The first name is for the output file.  By default, it is given
the extension ".COM".  The second name is for the map file
(default extension ".MAP") which gives the values of all the
global symbols.  ZLINK will always tell you how many global
symbols were undefined, but won't tell you what the undefined
symbols were unless you ask for a map file.  This does not
normally matter as ZRES will list all unresolved references.

All the names to the right of the '=' are input files, with the
default extension ".OBJ".  The first input file must have been
compiled WITHOUT the -M option.  Ordinarily, it will be the one
with main().  The other files can be mentioned in any order.


LIBRARY FILES

Several different groups of library files are included:

CTYPE           Character classification functions.
FLOAT           Floating point arithmetic routines.
GETOPT          Processing of command line options.
IOLIB           Basic input/output and integer arithmetic.
MATH            Mathematical routines.
PLOT            Plotting routines for the Amstrad PCW
PRINTF          Generic functions for output.
PRINTF1         Output routine _printf(), integer only.
PRINTF2         Output routine _printf() with floating point.
SCANF           Generic input routines.
SCANF1          Input routines for integers only.
SCANF2          Input routines for integers and floating point.
STRING          String handling routines.
WILDCARD        Expand wildcard filenames.


The IOLIB.OBJ library must be included in every executable.
The rest of the library is kept in the file CLIB.LIB and an
index to the library is kept in the file CLIB.IDX.  These files
are built using the utility ZLIB.

Library functions should normally be declared to the compiler by
including the appropriate header file.  For example, if floating
point operations are needed, the source file should contain:

        #include <stdio.h>
        #include "float.h"

        ...                     (rest of source code)

The header files are: stdio.h, string.h, math.h, ctype.h, plot.h
and float.h.

Compilation, assembly, and linking would consist of:

        A>cc0 alpha
        A>zmac alpha=alpha
        A>zres a: clib alpha
        A>submit clib

An optional stage in this process is to optimise the assembly
code produced by the compiler.  This is achieved by calling the
```



```
optimiser, ZOPT.COM:

        A>cc0 alpha
        A>zopt alpha
        A>zmac alpha=alpha
        A>zlink a: clib alpha
        A>submit clib

You may prefer to omit the optimisation until your program has
been debugged.  This will save some time during development.


SAMPLE COMPILATION

    M>cc0 test


    * * *  Small-C/Plus  Version 1.00  * * *

           Cain, Van Zandt, Hendrix, Yorston

           25th February 1988


    TEST.c                              <file names echoed>

    #include <stdio.h>                  <include files echoed>
    #end include
    #include <math.h>
    #end include
    #include "float.h"
    #end include
    ====== main()                       <function names echoed>
    ====== out()
    ====== alpha()
    ====== beta()
    ====== gamma()
    ====== putnum()
    ====== outf()

    Minimum bytes free: 4225
    Symbol table usage: 38
    There were 0 errors in compilation.

    M>zmac test=test
    SSD RELOCATING  (AND EVENTUALLY MACRO) Z80 ASSEMBLER VER 1.07

        0  ERRORS

    M>zres b: clib test
    bytes free: 13301
    Copying FLOAT                       <loading from library>
    Copying GETS
    Copying PRINTF
    Copying IFIX
    Copying SQRT
    Copying FLOOR
    Copying QFLOAT
    Copying PRINTF2
    Copying UTOI

    Symbol table   152/700              <ZRES statistics>
```

```
    Library table  137/500
    Index table    453/600

    M>submit clib
    M>zlink TEST=TEST,IOLIB,CLIB        <from submit file>
    SSD LINK EDITOR
          0 UNDEFINED SYMBOL(S).
    M>era clib.obj
    M>B:
    B>pip TEST.COM=m:
    B>era m:clib.sub
```

PERFORMANCE

The program test.c on this disk (with 156 lines) was compiled in 25.0 sec on a 4 MHz Z-80 (with interrupts turned off and working from RAM disk). This gives a compilation speed of approximately 6 lines/sec.

The following floating point benchmark (from Dr. Dobb's Journal, Mar 84) finished in 637 seconds on a 4 MHz Z-80, with the result 2500.01047:

```
        #include <stdio.h>
        #include <math.h>
        #include "float.h"
        int i;
        double a;
        main()
        {       a=1.0; i=1;
                printf("starting\n");
                while(i++<2500)
                        {a=tan(atan(exp(log(sqrt(a*a)))))+1.0;}
                printf("Result is %20.12f ",a);
        }
```

A Small-C/Plus translation of the whetstone benchmark produced a result of 2400 whetstones/second.

INTERNAL DOCUMENTATION

This is a recursive descent, one pass compiler producing assembly language. The two major changes that have been made to speed it up relative to Ron Cain's original compiler are a hash coded symbol table and 1K disk buffers. Also, the compiler source makes use of many of the new features which have been added during development. For example, symnbol table entries are now represented as structures.

Global symbols defined in a C program are prefixed by 'Q' so they do not conflict with any global symbols in libraries.

A compiled program has the following layout...

```
        100H to _end-1          program & data
        _end to *heaptop        heap
        *heaptop to *SP-1       unused
        *SP to *(0006)          stack
```

The symbol _end is defined by ZLINK at link time, and points to

the first byte above program and data. The variable heaptop (initialized to _end) always points to the first byte above the heap. The stack pointer register SP is initialized to the first location above user memory (pointed to by the word at 0006). The variable names _end and heaptop are visible only to assembly language programs, since they do not begin with Q's.

Assembly language functions can be called by C programs. A C function evaluates each of its arguments (left to right), pushes them onto the stack, then calls the named function. Therefore, the assembly language function should expect that the top word on the stack is the return address, the next word is the last argument, etc. If the function is to return an integer or character value, it should be left in the HL register. Double values should be left in the 6 byte area starting at FA.

The compiler will quite happily compile itself, though it is now so big that it must be optimised if it is to fit in memory. In addition, I have successfully ported it to a Unix system. (That's why there are all the #ifdef SMALL_C's.) Note that the #asm preprocessor directives should be removed, as the Unix preprocessor doesn't like them. The resulting cross-compiler cannot handle floating point constants. This is because the Small-C/Plus compiler assumes that the Z80 floating point routines are being used, while in a Unix environment the f.p. routines appropriate to the host system will be linked in.

WARNINGS

In addition to the limitations mentioned above, the user should be aware of the following...

Functions are assumed to return an integer unless the compiler is told otherwise. In making an explicit declaration of the type of object a function returns, you have to use two lines...

```
        double frodo();
        frodo(x,y,z) int x; double y,z;
```

You can't combine them, as in standard C...

```
        double frodo(x,y,z) ...         /* not accepted */
```

The declaration "double frodo();" must appear before any use of the function, or the compiler will assume while generating the calling code that the function returns an integer.

The floating point routines in the FLOAT library (though none of the code produced by the compiler) use several of the undocumented Z-80 op codes, so they may not work on some processors. FLOAT also uses the alternate register set.

The floating point routines do not meet the proposed IEEE standard.

Ronald M Yorston
1 Church Terrace
Lower Field Road
Reading
RG1 6AS

22nd May 1988

CTYPE Library Documentation

The functions in this library classify characters into one of a number of groups. They return true (non-zero) if the character belongs to the given group and false (zero) if it doesn't.

FUNCTIONS

isdigit(c) char c;
determines if c is a decimal digit (0-9)

isupper(c) char c;
determines if c is an uppercase letter (A-Z)

islower(c) char c;
determines if c is a lowercase letter (a-z)

isspace(c) char c;
determines if c is a white-space character (space, tab, vertical tab, carriage return, line feed or form feed)

ispunct(c) char c;
determines if c is a punctuation character (all ASCII codes except control codes or alphanumerics)

iscntrl(c) char c;
determines if c is a control character (ASCII codes 0-31 or 127)

isalnum(c) char c;
determines if c is an alphanumeric (A-Z, a-z or 0-9)

isxdigit(c) char c;
determines if c is a hexadecimal digit (0-9, A-Z or a-z)

isalpha(c) char c;
determines if c is alphabetic (A-Z or a-z)

isprint(c) char c;
determines if c is a printable character (ASCII codes 32-126)

isgraph(c) char c;
determines if c is a graphic symbol (ASCII codes 33-126)

isascii(c) char c;
determines if c is an ASCII character (0-127)

toupper(c) char c;
returns the uppercase equivalent of c if c is lowercase, otherwise it returns c unchanged

tolower(c) char c;
returns the uppercase equivalent of c if c is lowercase, otherwise it returns c unchanged

toascii(c) int c;
returns the ASCII equivalent of c, effectively just by masking with 0x7f

INTERNAL DOCUMENTATION

All the classification functions use a table with one entry for each ASCII character. An entry is also included for EOF. The argument of each classification funtion must be an ASCII character if valid results are to be obtained. Use toascii() if necessary.

FLOAT Library Documentation

FLOAT contains the floating point arithmetic routines, and some functions visible to the user's program.

GENERAL INFORMATION

These routines will execute only on a Z-80. They use the alternate registers and some of the undocumented instructions of that processor. They do not conform to the IEEE floating point standard. The routines were written by Neil Colvin, and are worth study. They are the best code I have ever seen for the Z-80. - Jim Van Zandt

FLOATING POINT FORMAT

Each floating point number is 6 bytes long, and consists of a 40 bit fraction (most significant byte in the highest address) and an 8 bit exponent. For nonzero numbers, the fraction f has a value in the range 0.5 <= f < 1.0. Since its most significant bit would always be 1, it would carry no information and is replaced by the sign bit (set for a negative number). The exponent is 80H if the number is in the range 0.5 <= x < 1.0, and is increased by 1 for each place the binary point of f should be moved to the right. For example:

Representation	Number
00h,00h,00h,00h,00h,80h	0.5
00h,00h,00h,00h,80h,80h	-0.5
00h,00h,00h,00h,00h,81h	1.0
00h,00h,00h,00h,00h,7fh	0.25
0fah,33h,0f3h,04h,035h,80h	sqrt(.5) = .707106...
38h,0a9h,0d8h,5bh,5eh,7fh	1/log(10) = .43429...
21h,0a2h,0dah,0fh,49h,81h	pi/2 = 1.5707...

ARITHMETIC OPERATIONS

Each of the primary operations (DADD, DSUB, DMUL, and DDIV) takes its first operand from the stack (under the return address) and the second from the fixed location FA (for Floating point Accumulator). The result of the operation is left in FA. For example, we have the following C expression and its translation into calls to floating point operations:

```
;double a,b,c,d;
;main()
QMAIN:
;{      a=b+c/d;
        LD HL,QB        ;get address of 1st operand
```

```
                CALL DLOAD      ;put operand in FA
                CALL DPUSH      ;move from FA to stack
                LD HL,QC        ;put 2nd operand...
                CALL DLOAD
                CALL DPUSH      ;...on stack
                LD HL,QD
                CALL DLOAD      ;put D in FA
                CALL DDIV       ;find c/d
                CALL DADD       ;find b+c/d
                LD HL,QA        ;load destination address
                CALL DSTORE     ;save result
        ;}
                RET

        QA:     DS 6            ;declare storage space
        QB:     DS 6
        QC:     DS 6
        QD:     DS 6


FUNCTIONS

Each of these functions return a double:

amin(x,y) double x,y;   returns the smaller of x, y
amax(x,y) double x,y;   returns the larger of x, y
float(x); double x;     integer to floating point conversion
fmod(x,y); double x,y;  mod(x,y)
                        if 0 < y then 0 <= mod(x,y) < y and
                        x = n*y + mod(x,y) for some integer n
fabs(x); double x;      absolute value
floor(x); double x;     largest integer not greater than
ceil(x); double x;      smallest integer not less than
rand();                 random number in range 0...1
seed(x); double x;      seed random number generator

This function returns an int:

int ifix(x); double x;  floating point to integer
                        (takes floor first)

The floating point package also includes the two constants:

double pi, halfpi ;     constant values of pi and pi/2
                GETOPT Library Documentation

GETOPT is a function used to process options from the command
line.


SYNOPSIS

        int getopt(argc, argv, optstring)
        int argc ;
        char **argv ;
        char *optstring ;

        extern char *optarg ;
        extern int optind ;

DESCRIPTION
        Getopt returns the next option letter in 'argv' that
```



```
        matches a letter in 'optstring'.  'Optstring' is a
        string of recognised option letters.  If a letter is
        followed by a colon the option is expected to have an
        argument which may or may not be separated from it by a
        white space.  'Optarg' is set to point to the start of
        the option argument on return from getopt.

        Getopt places in 'optind' the 'argv' index of the next
        argument to be processed.  Because 'optind' is external
        it is initialised to zero before the first call to
        getopt.

        When all the options have been processed (i.e. up to the
        first non-option argument), getopt returns EOF.  The
        special option -- may be used to delimit the end of the
        options.  EOF will be returned and -- will be skipped.

DIAGNOSTICS
        Getopt prints an error message on stderr and returns a
        question mark (?) when it encounters an option letter
        not included in 'optstring'.

EXAMPLE
        The following code fragment shows how one might process
        the arguments for a command which can take the mutually
        exclusive options 'a' and 'b', and the arguments 'f'
        and 'o', both of which require arguments:

          main(argc,argv)
          int argc ;
          char **argv ;
          {
             int c ;
             extern int optind ;
             extern char *optarg ;
             .
             .
             .
             while ((c=getopt(argc,argv,"abf:o:))!= EOF)
                switch (c) {
                case 'a' :
                   if (bflg)
                      errflg++ ;
                   else
                      aflg++ ;
                   break ;
                case 'b' :
                   if (aflg)
                      ++errflg ;
                   else
                      bproc() ;
                   break ;
                case 'f' :
                   ifile = optarg ;
                   break ;
                case 'o' :
                   ofile = optarg ;
                   break ;
                case '?' :
                default :
                   ++errflg ;
                   break ;
```

```
                }
             if (errflg) {
                fprintf(stderr, "Usage: ...") ;
                exit() ;
             }
             for (; optind < argc; ++optind) {
                .
                .
                .
             }
             .
             .
             .
          }

AUTHOR
        Original version by Henry Spencer, from Usenet.
        Translated for Small-C/Plus by Ronald M Yorston.
                IOLIB Library Documentation

IOLIB defines all the arithmetic operations, and the following
I/O, heap management, and error reporting functions:


FUNCTIONS

cpm(bc,de) int bc,de;
        The registers BC and DE are set to the values of the
        corresponding arguments, and a BDOS service request
        (CALL to 5) is made.  The value returned is the contents
        of A (sign extended).

getchar()
        Echoes and returns one character from the standard
        input, which is initially the keyboard but can be
        redirected from the command line.

putchar(c) char c;
        displays one character on the standard output, which is
        initially the console but can be redirected from the
        command line.  Adds LF after CR.  Returns c or EOF on
        error.

gets(buf) char buf[80];
        Gets a null-terminated string from the standard input.
        If I/O has not been redirected, the string comes from
        the keyboard and standard CP/M editing is permitted.  The
        maximum length of the string is fixed at 80 characters.
        The buffer buf should not be on the stack.

fgets(s,n,fd) char *s; int n; FILE *fd;
        Reads a line of input from the file fd into the
        character array s.  The newline is read and placed in
        the string.  The array is terminated with a null
        character.  At most n-1 characters will be read.  Fgets
        normally returns s, but returns 0 on end of file.

puts(s) char *s;
        Displays a null-terminated string on the standard
        output, using putchar().  Note that, unlike the Unix
        version, puts() does not add anewline at the end of the
        output.
```



```
fputs(s,fd) char *s, FILE *fd;
        Writes the null-terminated string pointed to by s to the
        file pointed to by fd.  Returns number of characters
        written, or -1 on error.

fopen(name,mode) char *name,*mode;
        Opens file "name".  "mode" is a pointer to a single
        character (either upper or lower case): "r" for read
        access, "w" for write access, and "a" for appending to
        an existing file.  fopen returns a file pointer (FILE *)
        which must be used for subsequent file accesses.
        For example...

        fd=fopen("frodo.c","r") opens FRODO.C for reading
        character=getc(fd)      gets a character from FRODO.C
        fd=fopen("sam.c","W")   opens SAM.C for writing
        putc(character,fd)      writes a character to SAM.C

        Up to five files may be open at once. (If more are
        needed, change NBUFS in IOLIB.C, compile, and assemble.)

fclose(fd) FILE *fd;
        Closes the file with file pointer fd.

getc(fd) FILE *fd;
        Returns the next character from the file (not sign
        extended), or -1 at end of file.  Line feeds are
        discarded, and control Z (1AH) signals end of file.

getb(fd) FILE *fd;
        Return next byte from file (not sign extended), without
        regard to its value, or -1 if at end of file.  (Use this
        one to read a COM file.)

putc(c,fd) char c; FILE *fd;
        Write character c to a file.  If it is a carriage
        return (\n), write a line feed as well.  Returns c.

putb(c,fd) char c; FILE *fd;
        Write byte c to a file, without special handling of
        carriage return.

fflush(fd) FILE *fd;
        Flush buffer for file fd (which must be an output
        file) to disk.  Called automatically by fclose().

unlink(c) char *c;
        Unlink (delete) the file whose name is pointed to by c.
        Returns 0 on success and -1 on failure.

abs(x) int x;
        Returns the absolute value of x.

alloc(n) int n;
        Returns a pointer to a block of n bytes of memory
        (no error checking).

free(ptr) char *ptr;
        ptr should be one of the pointers returned by alloc.
        That block AND ALL BLOCKS ALLOCATED SINCE THEN are
        returned to the heap.
```

```
avail()
        Returns the number of bytes of memory available for
        the heap AND THE STACK.  If you allocate all of it
        and write over the stack, you will cause trouble.
        The safe way to get a big buffer is as follows:
                size=avail()-300;
                where=alloc(size);
                        /* initialize if needed... */
                i=0; while(i<size) {where[i]=0;}

err(s) char *s;
        Prints "\nERROR" and the message pointed to by s on
        the console, and (if enabled during compilation)
        performs a walkback trace. For example, fopen uses
        the call:
                err("OUT OF DISK BUFFERS");
        The walkback trace lists the functions that have
        been called but have not yet returned, with the
        most recently called function first. Any functions
        compiled without the trace option (and all assembly
        language functions) simply don't appear in the list.


COMMAND LINE PROCESSING

IOLIB includes code to give the user access to the command
line and optionally redirects the standard input and/or output.
This code is called automatically when the program starts.

_setargs() parses the command line, which is a series of items
separated by one or more spaces.  A '<' followed by a file
name will redirect the standard input (used by getchar() and
gets()) to that file.  A '>' followed by a file name will
similarly redirect the standard output (used by putchar() and
puts()).  If the file exists, it will be silently deleted.  '>>'
followed by a file name will also redirect the standard output,
but if the file already exists then the new characters will be
appended to the existing data.  In any of the above cases, one
or more spaces can appear before the file name.  Items other
than the above are saved for later access as command line
arguments.  As far as argc and argv are concerned, I/O
redirection commands are invisible.  Regardless of I/O
redirection, err() will always display its message and walkback
trace (if any) on the console, putc(c,1) or putc(c,stderr) will
always send the character c to the console, and getc(0) will
always get a character from the keyboard.


POTENTIAL IMPROVEMENTS
        alloc() and free() should permit blocks of memory to be
        allocated and freed in any order.  (But see file malloc.c
        on this disk.)
NAME
        map2sym - convert .MAP files to .SYM

SYNOPSIS
        map2sym  file

DESCRIPTION

MAP2SYM takes as input a .MAP file as produced by ZLINK and
```



```
turns it into a .SYM file, suitable for use with the Z8E
debugger.  Only the file name should be specified on the command
line, the extension .MAP is assumed for the input file, and .SYM
for the output file.


AUTHOR
        Ron Yorston
                MATH Library User's Documentation

MATH supplies certain mathematical functions. They give
at least 11 significant digits in all known cases.


FUNCTIONS

sqrt(x);                square root
exp(x);                 exponential
log(x);                 natural logarithm
log10(x);               log base 10
pow(x,y);               pow(x,y) = x**y

atan(x);                arc tangent
                                (value in range -pi/2 to pi/2)
atan2(x,y);             arc tangent of x/y
                                (value in range -pi to pi)
cos(x);                 cosine
sin(x);                 sine
tan(x);                 tangent

cosh(x);                hyperbolic cosine
sinh(x);                hyperbolic sine
tanh(x);                hyperbolic tangent

In each case, the arguments and the value returned are doubles.


INTERNAL DOCUMENTATION

The square root is calculated by the Newton-Raphson method. The
hyperbolic functions are calculated in terms of the exponential.
The other functions are calculated by means of power series
after range reduction.
                    PLOT Library Documentation

The PLOT library contains routines to use the graphics
facilities of the Amstrad PCW computer.  The PLOT.RSX must be
loaded along with the compiled program if the plot() routine is
used.  This is done automatically by ZRES.


FUNCTIONS
plot(x,y,action) int x,y,action;
        The coordinates x and y refer to the Amstrad PCW screen.
        (0,0) is the pixel at the bottom left, (255,719) is the
        pixel at the top right.  The action variable specifies
        what will happen at that pixel.  The following values
        are allowed for action (defined in the plot.h header
        file):

        SET             turn on pixel
        RESET           turn off pixel
```

```
        TOGGLE          change state of pixel
        GETBIT          plot() returns bit
        GETBYTE         plot() returns byte containing pixel

cursor(row,column) int row,column;
        Position cursor at given row (0...31) and column
        (0...89).

viewport(top,left,height,width) int top,left,height,width;
        Set viewport.  'Top' is the line number of the top of
        the viewport.  'Left' is the column number of the left
        of the viewport.  'Height' is the number of rows in the
        viewport, and 'width' is the number of columns.

line(x1,y1,x2,y2,action) int x1,y1,x2,y2,action;
        Draw a line from (x1,y1) to (x2,y2).  The actions
        allowed are SET, RESET and TOGGLE.

box(x1,y1,x2,y2,action) int x1,y1,x2,y2,action;
        Draw a box with (x1,y1) at the top left and (x2,y2) at
        the bottom right.  The actions allowed are SET, RESET
        and TOGGLE.
                PRINTF Library Documentation

PRINTF2 supplies formatted output like that described by
Kernighan and Ritchie.  The input conversion routine utoi (for
unsigned integers) is also supplied.  PRINTF1 is identical
except that formats 'f' and 'e' of printf and functions ftoe and
ftoa are missing.  Thus, PRINTF2 requires FLOAT while PRINTF1
does not require it.  The generic functions are provided in the
library PRINTF, thus a given program using printf will require
PRINTF and one of PRINTF1 or PRINTF2.


FUNCTIONS

printf(controlstring, arg, arg, ...) -- formatted print
        "controlstring" is a string which can contain any of
        the following format codes:

                %d      decimal integer
                %u      unsigned decimal integer
                %x      hexidecimal integer
                %c      ASCII character
                %s      null-terminated ASCII string
                %f      fixed point conversion for double
                %e      floating point conversion for double

        For each format code, there is an "arg" - a pointer to
        an object of that type. Between the '%' and the format
        code letter field specification may appear. For formats
        'f' and 'e', the specification consists of two integers
        separated by a period. The first specifies the minimum
        field width, and the second the number of digits to be
        printed after the decimal point. For all other formats,
        the specification consists only of the one integer
        giving the minimum field width. If there is no field
        specification, the item is printed in no more space
        than is necessary.

                Example                 Output
```



```
        printf(" decimal: %d ",15+2)      decimal: 17
        printf(" unsigned: %u ",-1)       unsigned: 65535
        printf(" hexidecimal: %x ",-1)    hexidecimal: FFFF
        printf(" string: %s ","hello")    string: hello
        printf(" character: %c ",65)      character: A
        printf(" fixed: %f ",1./7.)       fixed: .142857
        printf(" exponent: %8.5e ",1./7.) exponent: 1.42857e-1

        printf returns the number of characters output.

fprintf(unit, controlstring, arg, arg, ..)
        Provides functions similar to printf, but with output
        going to the file associated with 'unit'.
        fprintf returns the number of characters output.

sprintf(string, controlstring, arg, arg, ..)
        Provides functions similar to printf, but with output
        going to the character pointer 'string'.
        sprintf returns the number of characters output.

itod(n, str, sz)  int n;  char str[];  int sz;
        convert n to signed decimal string of width sz,
        right adjusted, blank filled; returns str
        if sz > 0 terminate with null byte
        if sz = 0 find end of string
        if sz < 0 use last byte for data

itou(nbr, str, sz)  int nbr;  char str[];  int sz;
        convert nbr to unsigned decimal string of width sz,
        right adjusted, blank filled; returns str
        if sz > 0 terminate with null byte
        if sz = 0 find end of string
        if sz < 0 use last byte for data

itox(n, str, sz)  int n;  char str[];  int sz;
        converts n to hex string of length sz, right adjusted
        and blank filled, returns str
        if sz > 0 terminate with null byte
        if sz = 0 find end of string
        if sz < 0 use last byte for data

ftoa(x,f,str) double x; int f; char *str;
        converts x to fixed point string with f digits after
        the decimal point, return str

ftoe(x,f,str) double x; int f; char *str;
        converts x to floating point string with f digits after
        the decimal point, return str

utoi(decstr, nbr)  char *decstr;  int *nbr;
        converts unsigned decimal ASCII string to integer
        number. Returns field size, else ERR on error. (This is
        used to interpret the specification fields.)


AUTHOR
        J. E. Hendrix for the original routines. J. R.
        Van Zandt for ftoa, ftoe, and the floating point
        modifications in printf.


INTERNAL DOCUMENTATION
```

The method used in ftoa to convert to a decimal string involves more divisions than the classical method, but does not require that the original number be scaled down at the beginning. It was found that this initial scaling was causing loss of precision. The present algorithm should always convert an integer exactly if it can be represented exactly as a floating point number (that is, if it is less than 2**40).

The routines seen by the user (printf, fprintf, sprintf) are in the library PRINTF. The routine which does all the hard work, _printf, is in either PRINTF1 or PRINTF2 depending on whether or not floating point output is required.

SCANF Library Documentation

SCANF2 supplies formatted input like that described by Kernighan and Ritchie. SCANF1 is identical except that formats 'f' and 'e' are missing. Thus, SCANF2 requires FLOAT while SCANF1 does not. The generic functions which the user actually calls are provided in the library SCANF, thus a given program using scanf will require SCANF and one of SCANF1 or SCANF2 .

FUNCTIONS

scanf(controlstring, arg, arg, ...)
scanf reads a series of white-space-separated fields from the standard input, converts them to internal format according to a specification in the 'control-string', and stores them at the locations indicated by the arguments. It returns a count of the number of fields converted or EOF if no fields were processed because end-of-file was reached.

'controlstring' can contain any of the following format codes to tell scanf how to treat each field:

%d	decimal integer
%u	unsigned decimal integer
%x	hexadecimal integer (ignore 0x or 0X)
%o	octal integer
%c	character (no skip over white space)
%s	null-terminated ASCII string
%e	floating point conversion for double
%f	floating point conversion for double

For each format code, there is an 'arg' - a pointer to an object of that type. Between the '%' and the format code letter an asterisk and/or a decimal integer constant may appear. An asterisk indicates that the corresponding field in the input should be skipped. No 'arg' is necessary in this case. A number indicates the maximum field width in characters. Conversion terminates either when the given number of characters is exhausted or when a white space character is found.

Example:

```
int i ;
double x ;
char name[50] ;
scanf("%2d %f %*d %2s", &i, &x, name);
```


with input

```
56789 0123 45a72
```

will assign 56 to i, assign 789.0 to x, skip over 0123, and place the string 45 in name. The next call to any input routine will start searching at the letter a.

fscanf(unit, controlstring, arg, arg,...)
Provides functions similar to scanf, but with input taken from the file associated with 'unit'. fscanf returns the number of fields successfully converted.

sscanf(string, controlstring, arg, arg,...)
Provides functions similar to scanf, but with input taken from string pointed to by the character pointer 'string'. sscanf returns the number of fields successfully converted.

atof(str) char *str;
converts from ASCII to floating point, returns the double value. The general input format is [-][integer][.[fraction]][e[-]exponent], where things in brackets are optional (except that either an integer or a fractional part must be present).

Examples				Values
1	1.	1.0		1.
.1	1.e-1	10.e-2	.01e1	0.1

Conversion stops with the first character that doesn't match the above format.

AUTHOR
J. E. Hendrix for the original routines, with some modifications and generalisations by R M Yorston.

INTERNAL DOCUMENTATION

The routines seen by the user (scanf, fscanf, sscanf) are in the library SCANF. The routine which does all the hard work, _scanf, is in either SCANF1 or SCANF2 depending on whether floating point output is required or not.

STRING Library Documentation

The STRING library contains functions to deal with null-terminated character strings. These functions are similar to their Unix counterparts.

FUNCTIONS

strlen(str) char *str;
returns a count of the number of characters in the string 'str'. The null character at the end of the string is not included in the count.

strcat(dest, sour) char *dest, *sour;
appends the string at 'sour' to the end of the string at 'dest'. A null character terminates the end of the new

```
        'dest' string.  The space reserved for 'dest' must be
        long enough to hold the resulting string.  The function
        returns 'dest'.

strncat(dest, sour, n) char *dest, *sour; int n;
        works like strcat(), except that a maximum of n
        characters from the source string are transferred to the
        destination.

strcmp(str1, str2) char *str1, *str2 ;
        returns an integer less than, equal to, or greater than
        zero depending on whether the string at 'str1' is less
        than, equal to, or greater than the string at 'str2'.

strncmp(str1, str2, n) char *str1, *str2; int n;
        works like strcmp, except that a maximum of n characters
        are compared.

strcpy(dest, sour) char *dest, *sour;
        copies the string at 'sour' to 'dest'.

strncpy(dest, sour, n) char *dest, *sour; int n;
        works like strcpy(), except that n characters are
        transferred regardless of the length of the source
        string.  If the source string is too long it is
        truncated and a null character is placed at the end of
        the destination string.

strchr(str, c) char *str, c;
        returns a pointer to the first occurrence of the
        character 'c' in the string at 'str'.  A NULL pointer is
        returned if the character is not found.

strrchr(str, c) char *str, c;
        works like strchr(), except that the rightmost
        occurrence of the character is searched for.
TRACE

If the trace option of the compiler is used, each call to err()
results in a walkback trace of function calls.  (Err() is in the
IOLIB library. For details, see IOLIB.DOC.)

A header and two calls are added to the code generated for each
function. The function header contains a string with the function
name.

        ;trials()
        CC4:    DB 'trials',0   ;the function name
        QTRIALS:
                LD HL,CC4       ;save pointer to function
                PUSH HL         ;header block.
                CALL CCREGIS    ;register function entry.
                                ;ccregis() pushes onto the stack
                                ;a pointer to the function that
                                ;called this one, and saves in
                                ;CURRENT a pointer to this one.
        ;{      z=a(x);
                LD HL,QX

                ...             ;regular code.

        ;}
```



```
                CALL CCLEAVI    ;register function return
                                ;(resets CURRENT to point to
                                ;the function that called this
                                ;one)
                POP BC          ;discard the pointer added by
                                ;ccregis().
                POP BC          ;discard the pointer to the
                                ;header block of this function.


                RET

Note that this method permits a walkback trace even in the
presence of recursive function calls.
                WILDCARD Library Documentation


SYNOPSIS
        wildcard(spec)
        char *spec;

DESCRIPTION
        The function wildcard() returns a pointer to an array of
        pointers to character strings.  In full C that would be
        written:

            char **wildcard(spec)
            char *spec;

        In Small-C/Plus the best we can do is return an integer.
        The character strings contain file names from the current
        disk which match a wildcard specification at 'spec'.  Up
        to 63 pointers are available, with the last valid one
        being followed by NULL pointer.  If no files match the
        specification wildcard() returns a NULL pointer.

NOTE
        The file specification must be twelve characters long in
        the format "FFFFFFFF.EEE".  Names should be padded at
        the right with spaces to fit this format.  A wild
        character is denoted by a question mark.

AUTHOR
        Ronald M Yorston
NAME
        zlib - creates library file

SYNOPSIS
        zlib    libfile

DESCRIPTION

ZLIB goes through all files with the extension "OBJ" on the
current disk and builds a library file and index file from them.
The index is used by the ZRES program to resolve references to
library routines.

The 'libfile' should not have an extension.  Two output files
are created, LIBFILE.LIB is the library and LIBFILE.IDX is the
index.

The index file has two parts.  In the first part each module in
```

```
the library is listed together with its offset in the library
file (sector number and offset within sector).  In the second
part the labels in each module are listed.  If a label is
referenced in a module it is followed by the negative of the
module number.  If a label is defined in a module it is followed
by the module number.

The library file contains all the .OBJ files as separate
modules.


EXAMPLE

To index all the "OBJ" files on drive C:, with the output being
sent to the files 'clib.lib' and 'clib.idx', use the command:

        C>zlib clib
        C>



AUTHOR
        Ron Yorston
NAME
        zlink - linkage editor

SYNOPSIS
        zlink  comfile,mapfile=relfile,relfile...

DESCRIPTION

ZLINK is a linkage editor for programs assembled by ZMAC.
"Comfile" is the executable output file, with default extension
"COM". "mapfile" is a listing of global symbols and their
values, with default extension "MAP". The "relfiles" are the
input files, with default extension "OBJ". The command line may
be up to 128 bytes long. If a longer list of input files is
needed, an "&" may be appended to the last name and ZLINK will
prompt for more input with "&LINK>".

The output files are both optional, so that

        zlink sam=sam

reads SAM.OBJ and creates SAM.COM, while

        zlink ,sam=sam

creates only SAM.MAP, and

        zlink sam,sam=sam

creates both. The map file is very convenient for reference
while using a debugger. The destinations "CON:" and "LST:"
are also legal for the map file. The way to find out what
symbols are imported and exported by an object file is to
execute the linker and request only the map file:

        zlink ,con:=bilbo

One of the last symbols shown will be "_END". Subtract 100H
from its value to get the length of the executable code.
```



```
If ZLINK is called with no arguments, it will obey multiple
commands of the above format, prompting for each with "LINK>".
An empty command line terminates the input.

ZLINK defines the symbol _END to point to the first byte after
the program (including all code and data).

Data areas with contents otherwise unspecified are initialized
to zero.


EXAMPLE

ZLINK links itself as follows (though sources are not included
here):

        C>zlink zl,zl=zlink,linkp1,linkp2,linkproc,&
        SSD LINKAGE EDITOR VERSION 1.4
        &LINK>mfsp,fsparse,gfspecs,linksadd,linkio,&

        &LINK>outmap,wrtrel,link01,link02,linkram

              0 UNDEFINED SYMBOL(S).
        C>


BUGS

If one input file comes from an assembly language file with an
AORG directive, then the next input file doesn't correctly
import global symbols.

Undefined symbols aren't listed unless a MAP file was
requested.


NOTE

ZLINK accepts .OBJ files as produced by ZMAC.  An additional
patch has been added to handle the CLIB.OBJ file produced by
ZRES.  As a result of this patch, the end of a module may be
marked by a record of the form

        DB      2,1

In this case the linker continues to read the next module from
the same file instead of opening a new file.  This allows
multiple moudules to be held in the CLIB.OBJ file.  (Patch by
R M Yorston.)

AUTHOR
        Bruce Mallett
NAME
        zmac - relocating Z-80 assembler

SYNOPSIS
        zmac  relfile,listfile=asmfile

DESCRIPTION

ZMAC is a Zilog mnemonic assembler with command and language
```

```
syntax similar to DEC assemblers. "relfile" is the object file
name, with the default extension ".OBJ" (for the format, see
OBJ.DOC). "listfile" is the listing file, with the default
extension ".PRN". In addition to standard disk files, you can
specify "LST:" for the list device or "CON:" for the console.
"asmfile" is the input assembly language file, with the default
extension ".ASM". The output files are both optional, so that

        zmac frodo=frodo

reads FRODO.ASM and creates FRODO.OBJ, while

        zmac ,frodo=frodo

creates only FRODO.PRN, and

        zmac frodo,frodo=frodo

creates both. Listing files are rarely needed except for final
documentation, since lines with syntax errors are automatically
listed to the console.

If ZMAC is called with no arguments, it will obey multiple
commands of the above format, prompting for each with "ZMAC>".
Operating this way saves time, since the assembler gets read in
only once. An empty command line terminates the input.


INPUT LANGUAGE

The language accepted by ZMAC is like that for the Zilog
assembler, with a few exceptions...

ZMAC does not require the "-$" after relative jump arguments.
The standard and ZMAC syntaxes are as follows:

standard:               JR      SOMEWHERE-$
ZMAC:                   JR      SOMEWHERE .

For equates, the syntaxes are:

standard:       BELL    EQU     7H
ZMAC:           BELL    =       7H .

A colon is forbidden after an equated symbol, but both a colon
and whitespace (space, tab, or carriage return) are required
after a label.

Symbols defined in the current module which are to be
referenced in other modules (exported symbols), or those
referenced in the current module but defined elsewhere
(imported symbols) must be declared GLOBAL:

                        GLOBAL  VAR

The ORG directive is illegal. There is instead the AORG
("absolute ORG") to set the program counter to a given absolute
address. The bad news is that ZLINK has a bug in its handling
of AORG. If one module has an AORG, then the NEXT module can't
correctly import symbols. The good news is that an AORG is
hardly ever necessary. ZLINK starts the code at 100H by
default. There is also an RORG ("relative ORG") directive,
```



```
which sets the program counter to a particular value with
respect to the module beginning.

Symbols and opcodes can be in either upper or lower case (no
case distinction). A symbol may have at least 100 characters,
and the first 16 characters are significant. In addition to the
standard alphabetic and numeric characters, the four characters
"_$.%" are also permitted in symbols. A "$" by itself stands
for the value of the program counter (the location of the first
byte in the CURRENT machine instruction). For example, an
infinite loop can be coded as "JP $".

Numbers should start with a numeral, which can be zero. By
default, the number is interpreted in decimal. The base of the
number can be set by a letter at the end of the number: D for
decimal, H for hex, O for octal, or B for binary.

The assembler can evaluate quite complex expressions.
Multiplication and division have higher precedence than
addition or subtraction (as usual for most software, but untrue
for the Zilog assembler). Parentheses are permitted to enforce
a certain evaluation order, but parentheses around an entire
expression denote indexing.

The unary operations are:       +    (no operation)
                                -    negate (2's complement)
                                #    1's complement

The binary operations are:      + - * / as usual
                                \       inclusive or
                                &       and

EXAMPLE

Consider the following assembly:

        C>zmac demo,demo=demo
        SSD RELOCATING   (AND EVENTUALLY MACRO)  Z80 ASSEMBLER VER 1.07

            0  ERRORS

...or the equivalent assembly using interactive input:

        C>zmac
        SSD RELOCATING   (AND EVENTUALLY MACRO)  Z80 ASSEMBLER VER 1.07
        ZMAC>demo,demo=demo


            0  ERRORS
        ZMAC>


            0  ERRORS TOTAL
        C>

The resulting listing file DEMO.PRN is as follows:

                                                PAGE NO.    1
                    1 ;Demonstration of ZMAC assembly language
                    2 ;syntax and resulting object code
                    3 ;
                    4 ;declare imported symbol before use
```

```
                     5          GLOBAL  OMICRON
                     6 ;declare exported symbol before definition
                     7          GLOBAL  ALPHA
                     8 ;
                     9 ;Equal sign rather than "EQU",
                    10 ;and colon is illegal
   0001=            11 ONE      =       1
                    12 ;using local symbol
  '0000  0700'      13          DW      SIGMA
                    14 ;lower case is synonymous
  '0002  0700'      15          dw      sigma
  '0004  0C00'      16          DW      MU
                    17 ;both colon and whitespace (blank, tab,

                    18 ;or CRLF) are required after label
  '0006: 00         19 ALPHA:   DB      0
  '0007:            20 SIGMA:
                    21 ;using "extended alphabet"
                    22 ;in symbol names
  '0007: 01         23 _BETA:   DB      1
  '0008: 0F         24 BE_TA:   DB      15
  '0009: 02         25 .GAMMA:  DB      2
  '000A: 03         26 $DELTA:  DB      3
  '000B: 04         27 %EPSILON: DB     4
                    28 ;        "EF" is optional
  '000C: 05         29 MU:      DEFB    5
  '000D: 0600       30 NU:      DEFW    6
                    31 ;
  '000F:            32 RHO:     DS      16
                    33 ;precedence used in
                    34 ;evaluating expressions
  '001F  07         35          DB      1+2*3
  '0020? 0000       36          DW      OMICRON
  '0022  88         37          DB      88H
                    38 ;single or double quotes around string
                    39 ;(double either to insert into string)
  '0023  4A6F6527   40          DB      'Joe''s mom'
  '002C  20226861   41          db      " ""hates"" chocolate"
  '003E? 00         42          DB      OMEGA
  '003F  88         43          DB      88H
                    44 ;declare exported symbol after definition
                    45          GLOBAL  RHO
                    46 ;declare imported symbol after use
                    47          GLOBAL  OMEGA
    0  ERRORS
```

PAGE NO. 1

Addresses and data values subject to relocation are marked with single quotes. Imported values are marked with question marks.

FORMAT OF .OBJ FILE

The following information was gleaned from inspection of the source code of the assembler and linker, and output generated by the assembler. It didn't come from Bruce Mallett, so any errors aren't his fault. - Jim Van Zandt

The relocatable file created by ZMAC consists of a module record, a series of data records, symbol records, and set address records, and is terminated by an end of module record.

An end of module record has the format:

```
        DB      2,0
```

A module start record has the format:

```
LGH1:   DB      NEXT1-LGH1      ;# bytes in record
        DB      1               ;signals MODULE record
        DB      YY              ;descriptor bits (see below)
        DB      'FREEMONT'      ;optional module name
NEXT1:
```

A set address record is generated for each DEFS or DS opcode. It has the effect of resetting the linker's program counter. It has the format:

```
LGH2:   DB      NEXT2-LGH2      ;# bytes in record
        DB      2               ;signals SET ADDRESS record
        DB      YY              ;descriptor bits
        DW      XXXX            ;new value for program
;                               counter
NEXT2:
```

A data record has the following format:

```
LGH3:   DB      NEXT3-LGH3      ;# bytes in record
        DB      3               ;signals DATA record
        DS      28              ;one bit is set for each word
                                ;of data requiring relocation.
        DB      23,34,17,...,1BH ;1 to 224 bytes of data.
NEXT3:
```

A symbol record is used to import or export a global symbol. It has the format:

```
LGH4:   DB      NEXT4-LGH4      ;# bytes in record
        DB      4               ;signals SYMBOL record
        DB      YY              ;descriptor bits
        DW      XXXX    ;if defined here, XXXX is the value of
;                       the symbol. If not defined here, XXXX
;                       is the  address requiring the symbol.
;                       The value of the symbol will be added
;                       to the word  at XXXX. In either case,
;                       if "relocatable",  then XXXX is with
;                       respect to the beginning  of the module.
        DB      'GANDALF'       ;the symbol
NEXT4:
```

In the above records, the "descriptor bits" are defined as follows:

```
        bit 0   if word rather than byte
        bit 1   if defined here
        bit 2   if global rather than local
        bit 3   if relocatable rather than absolute
        bit 4   if value of symbol is to be shifted left
```

```
                by 3 bits.

The "shift left 3 bits" note is used when the bit number in a
SET, BIT, or RES instruction is an imported symbol. In those
instructions, the bit number field is in bits 3 through 5 of a
byte. Note that it is always characteristic of a use, never a
definition, of a symbol.

The object code corresponding to the above assembly listing is:

C>dump demo.obj
DUMP version 00.05
        RECORD:  0
0000 0301 002D 03A8 0000-0000 0000 0000 0000  ...-.(..........
0010 0000 0000 0000 0000-0000 0000 0000 0000  ................
0020 0007 0007 000C 0000-010F 0203 0405 0600  ................
0030 0502 0A1F 000C 0405-2000 4F4D 4943 524F  ........ .OMICRO
0040 4E0A 0404 3E00 4F4D-4547 413F 0300 0000  N...>.OMEGA?....
0050 0000 0000 0000 0000-0000 0000 0000 0000  ................
0060 0000 0000 0000 0000-0007 0000 884A 6F65  .............Joe
0070 2773 206D 6F6D 2022-6861 7465 7322 2063  's mom "hates" c
        RECORD:  1
0080 686F 636F 6C61 7465-0088 0B04 0B0A 0024  hocolate.......$
0090 4445 4C54 4108 0402-0100 4F4E 4508 040F  DELTA.....ONE...
00A0 0F00 5248 4F0B 040B-0900 2E47 414D 4D41  ..RHO......GAMMA
00B0 0D04 0B0B 0025 4550-5349 4C4F 4E0A 040B  .....%EPSILON...
00C0 0700 5349 474D 410A-040F 0600 414C 5048  ..SIGMA.....ALPH
00D0 410A 040B 0800 4245-5F54 4107 040B 0C00  A.....BE_TA.....
00E0 4D55 0704 0B0D 004E-550A 040B 0700 5F42  MU.....NU....._B
00F0 4554 4102 0000 0000-0000 0000 0000 0000  ETA.............

The first byte of relocation bits in the first data record
(relative address 0005 in the file) is A8 hex, or 10101000
binary, signifying that words beginning at bytes 0, 2, and 4
among the following data bytes must be relocated. The last nine
bytes displayed are extraneous, since the end of module record
is at 00F3 and 00F4.

For more information, see the source files.


POTENTIAL IMPROVEMENTS

Handle multiple program counters, such as one each for "code",
"initialized data", and "uninitialized data".

Permit "EQU" as well as "=".

Make colons optional after either equated symbols or labels.

Make "ORG" a synonym for "AORG".


BUGS

A file name can't include a '-'.


AUTHOR
        Bruce Mallett
NAME
        zopt - assembly code optimiser
```



```
SYNOPSIS
        zopt  [-c] asmfile

DESCRIPTION

ZOPT is an optimiser for the assembler output of the
Small-C/Plus compiler.  "asmfile" is a file containing the
output from the compiler.  An extension of "ASM" is assumed.
Five passes are made through the code making a variety of
optimisations.  The optimised code is left in the original file,
i.e. the original contents of that file are overwritten.  At the
end of the optimisation the program reports the savings it
manages to make on the console.

The -c flag instructs the optimiser to generate 'compact' code.
This means that certain in-line expansion of run-time routines
is not performed, resulting in smaller, though slightly slower,
code.  By default all optimisations are performed.


EXAMPLE

To optimise the file FRED.ASM use a command of the form:

        A>zopt fred
                ...
                optimisation statistics
                ...
        A>


AUTHOR
        Ron Yorston
NAME
        zres - resolve library references

SYNOPSIS
        zres  d: library objfile1 [objfile2 ...]

DESCRIPTION

ZRES resolves library references for a given object file or
list of object files.  To do this it must be provided with a
library file and index, as created by the ZLIB utility.

At the end of its processing ZRES generates a submit file which
will invoke the linker to construct an executable file with the
same name as the first "OBJ" file, but with the extension "COM".
The submit file also contains commands to: load a plot RSX (if
necessary), return control to drive "d:" and move the "COM" file
to that drive.  It also deletes temporary files.

It is intended that ZRES should be used in the CC.SUB script,
which moves from the current drive to the memory drive before
carrying out the compilation.

EXAMPLE

ZRES will resolve references for the file TEST.C on this disk as
follows:
```

```
        M>zres a: clib test


AUTHOR
        Ronald M Yorston
```