Micro Technology Unlimited

280200 MTU-C

USER MANUAL

OCTOBER, 1982

REV. A

TABLE OF CONTENTS

MTU C MANUAL ADDENDUM

INTRODUCTION

This manual is divided into two sections, the addendum and the AZTEC C manual. This addendum is intended to be used in conjunction with the AZTEC C manual provided by MANX Software Systems which follows, and the reference book, The C Programming Language by Kernighan and Ritchie. This addendum will provide additional information concerning implementation and usage details which apply to the MTU-130. Also, this addendum gives documentation for the MTUGRAPH.L and MTUMISC.L Libraries which were added by MTU.

MTU C DISTRIBUTION FILES

The following is a list of the files associated with the MTU-C Language System:

MAIN PROGRAMS	CCI.C	Compiler
	ASI.C	Pseudo Code Assembler
	AS65.C	6502 Assembler
	LN.C	Linker
UTILITY PROGRAMS	MKLIB.C	Library Maker
	ARCH.C	Archive Maker
	CMP.C	File Compare
	SIZE.C	Display File Size
	OD.C	File Dump
	NM.C	Display File Symbol Table
LIBRARY FILES	MTULIB.L	Standard Library
	MTUFLT.L	Floating Point Utilities Library
	MTUGRAPH.L	Graphics Library
	MTUMISC.L	Library of misc. functions
SOURCE FILES	STDIOSRC.L	Archive of Standard I/O Source
	PROGSRC.L	Archive of Source for CMP, SIZE, OD, and NM
	GRAPHSRC.L	Archive of Graphics Source
	MISCSRC.L	Archive of Source of misc. functions
	STDIO.H	Standard I/O Definitions file
	HELLO.T	HELLO source program
	GRDEMO.T	Graphics Demo source program

The following are brief descriptions of the purpose of each of these files. The files called "archive" files contain multiple files that were combined using ARCH.C.

CCI.C - This executable program file is the C compiler. It takes your C source program and compiles it into a source pseudo code file.

ASI.C - This executable program file is the Pseudo Code Assembler. It assembles the file generated by the CCI program into a relocatable object file.

AS65.C - This executable program file is a 6502 Assembler for MTU-C. It assembles 6502 source files into relocatable object files.

LN.C - This executable program file is the Linker program. It links the specified group of relocatable object files along with any specified library files into an executable object file. Such a file is executed the same as you would any user or utility program. You simply enter its name.

MKLIB.C -	This is an executable utility program which can combine relocatable object files into a library file. The library files are constructed in a manner which provides for efficient searching by the Linker (LN.C) program.
ARCH.C -	This is an executable utility program which can concatenate a group of files into a single file. It also can extract a copy, or delete one of the files which had been concatenated into the archive file.
CMP.C -	This is an executable utility program which compares two files on a byte by byte basis.
OD.C -	This is an executable utility program which displays the contents of a file in hex and ASCII.
SIZE.C -	This is an executable utility program which computes the sizes of the various parts of a relocatable object file.
NM.C -	This is an executable utility program which prints the symbol table contained in relocatable object files. It also displays the value and type for each of the symbols.
MTULIB.L -	This is a library file which contains all of the standard routines except for the floating point routines. This includes the routines which make up the pseudo code interpreter. It must be specified as the last file in the argument list for the LN.C program.
MTUFLT.L -	This is a library file which contains the floating point routines. This file must be included in the argument list for the LN.C program if your C program makes use of floating point values.
MTUGRAPH.L -	This is a library file which contains a set of graphics functions. It has functions equivalent to those commands found in the IGL.Z BASIC Library.
MTUMISC.L -	This is a library file which contains some miscellaneous functions.
STDIOSRC.L -	This is an archive file which contains the source files for the Standard I/O routines.
GRAPHSRC.L -	This is an archive file which contains the source files for the MTUGRAPH.L library.
MISCSRC.L -	This is an archive file which contains the source files for the MTUMISC.L library.
STDIO.H -	This is a C source file which contains definitions that are required when you use the standard I/O functions. If your C program uses any of the standard I/O routines, you must include the statement '#include "STDIO.H"' in your program so that the STDIO.H file will be read during compilation.
PROGSRC.T -	This is an archive file (created by the ARCH program) which contains the source files for the CMP.C, OD.C, SIZE.C, and NM.C programs.
HELLO.T -	This is a C source file for a very simple C demonstration program.
GRDEMO.T -	This is a C source file for a C demo program which provides examples on how to use most of the functions in the MTUGRAPH.L and MTUMISC.L libraries.

This section assumes you are already familiar enough with the C language to be writing C programs. If you are not, you should study Kernighan and Ritchie's manual first. For the MTU C Language System, there are a number of steps involved in creating an executable C program. Described in simplest terms, these steps are:

1. Create or edit your C source program using the MTU Editor (EDIT.C).

2. Compile your source program using the C Compiler (CCI.C).

3. Assemble the generated pseudo code source file using the Pseudo-Code Assembler (ASI.C). For 6502 assembly language source files, use the 6502 Assembler (AS65.C) instead.

4. Link the required relocatable object files and libraries into an executable file using the C Linker (LN.C).

To help aquaint you with this process, there is a source file called HELLO.T supplied on your distribution disk. This is a very simple program that prints "Hello world!" on the display, which already satisfies step 1 above. To execute step 2, you would enter the command:

```
CCI HELLO.T
```

This will compile the HELLO.T program into source pseudo code which is written to the file HELLO.I. If a specific output file is not given, as in the case above, the compiler will use the same name as the source file but with a ".I" extension. When the compiler has finished, you would execute step 3 by entering the command:

```
ASI HELLO.I
```

This will take the HELLO.I file generated by the compiler and assemble it into a relocatable object module which is written to the file, HELLO.R. When no output file is specified with the ASI program, it will use the same name as the source file but with a ".R" extension. Now you are ready to execute step 4, the last step. Since the HELLO.T program didn't use any of the special functions in the MTUFLT.L, MTUGRAPH.L, or MTUMISC.L libraries, you need to link HELLO.R to only the required MTULIB.L. The command would be:

```
LN HELLO.R MTULIB.L
```

This will link the relocatable object module, HELLO.R, to the required routines in MTULIB.L to form an independently executable file called HELLO.C. When no output file is specified, the Linker will use the same name as the first object module but with a ".C" extension. When the Linker has finished (about 2 minutes), the program is ready to run. Simply enter the command:

```
HELLO
```

and "Hello world!" should be printed on the display.

There is another C source program supplied on the C Distribution disk called GRDEMO.T. This is a program which gives examples of how to use many of the routines in the MTUGRAPH.L and MTUMISC.L libraries, which were developed by MTU. The commands for steps 2 and 3 are:

```
    CCI GRDEMO.T
    ASI GRDEMO.I
```

You will notice that it takes longer to compile and assemble this program than the HELLO program. The time is roughly 3 minutes to compile, and 2.5 minutes to assemble. Initially, it may appear that the ASI program has crashed because of the lack of disk activity. However, this is not the case. The ASI program uses a fairly large input buffer, so it has to access the source file relatively infrequently. Since the GRDEMO program uses some of the routines found in MTUGRAPH.L and MTUMISC.L, they must be specified in the argument list for the Linker. The command would be:

```
    LN GRDEMO.R MTUGRAPH.L MTUMISC.L MTULIB.L
```

When the Linker has finished, you may run the demo by entering the command:

```
    GRDEMO
```

This demo will clear the display and present legends for drawing a box with and without some simple shading, drawing with the light pen, and going into a sketch pad menu. In the sketch pad menu, you are able to draw using the GRIN cursor and the function keys.

THE C I/O SUBSYSTEM

The MTU-C Language System contains its own I/O subsystem which interfaces the C programs to CODOS. This I/O subsystem is implemented as a group of functions which will be linked to your program automatically. These routines provide a simple means of controlling your data files as well as the valuable feature of being able to do buffered I/O with the buffer size of your choice.

The I/O routines in the I/O subsystem are implemented on two levels. The first level contains the System I/O functions. These are functions which interface the I/O subsystem to the CODOS operating system. The second level of I/O functions contains the Standard I/O functions. These are the functions which allow buffered file access and are the ones you will typically use in your C programs. These will call upon the System I/O functions to perform I/O with the CODOS operating system.

The System I/O functions typically use an argument called the file descriptor which is very similar to CODOS channels. The file or device acted upon by the System I/O function will be the one "assigned" to the specified file descriptor. Associated with each descriptor is a mode which indicates if reading, writing, or both are permitted with that descriptor. The System I/O functions support 9 file descriptors, numbered 0 through 8. Also associated with each file descriptor is a CODOS channel to be used for accessing the assigned file or device. The CODOS channel will be equal to the file descriptor plus 1. The System I/O routines do not allow you to pick which file descriptor to use when opening a file or device. Instead, one is picked for you by the open() and creat() functions whose returned value is the chosen file descriptor. You can, however, count on the lowest unused descriptor to be chosen.

The Standard I/O functions use a stream argument to select the file or device upon which to act. This stream argument is actually a pointer to a set of parameters which are used by the functions to perform their I/O. These parameters contain information about the location, size, and current state of the buffer to be used while accessing the file or device. In addition, one of these parameters is, quite naturally, the file descriptor which is assigned to the file or device for this stream. The Standard I/O routines provide 8 streams which may be in use at one time. As with the System I/O routines, the Standard I/O routines do not let you pick which stream to use when opening the file or device. The fopen() function picks one for you and returns it as the function's value.

When a C program is executed, it is the task of the I/O subsystem to perform some initialization before the "main" routine begins executing. This initialization includes automatically opening the streams stdin, stdout, and stderr, plus performing any redirection of stdin and stdout that is called for by arguments in the command line. This initialization process has been optimized for the CODOS environment. The sequence of initialization performed by the I/O subsystem is as follows:

1. CODOS channels 4 through 9 are freed.

2. CODOS channels 1, 2, and 3 are checked for being assigned. If the channel is assigned, the associated file descriptor is marked as being permanently open. If the channel is not assigned, the associated file descriptor is opened for reading and writing to the console device. This channel will not be marked as permanently open.

3. The stdin stream is set to use file descriptor 0 (i.e. CODOS channel 1). The stdout stream is set to use file descriptor 1 (i.e. CODOS channel 2). The stderr stream is set to use file descriptor 2 (i.e. CODOS channel 3).

The significance of the file descriptor being permanently open is that closing that file descriptor will have no effect. This is necessary to allow job files to execute C programs, because the I/O subsystem will close all open streams and file descriptors when the program terminates. By allowing file descriptors 0, 1, and 2 to be made permanently open, they will remain assigned to the necessary files during execution of the job file. Otherwise they would be freed when the first C program terminates. There is no standard way of reassigning a permanently open file descriptor to another file. You may, however, reassign the stdin, stdout, or stderr streams using the freopen() function. The reassigned stream will use one of the other file descriptors, leaving the permanently open one unchanged.

Typically only CODOS channels 1 and 2 would be assigned when executing a C program. This would cause file descriptors 0 and 1 to be permanently open. File descriptor 2 would be assigned initially to the console, but would not be marked permanently open. Closing this file descriptor would free the channel and allow it to be reassigned to some other file or device.

If you want, you may specify in the command line what the stdin and stdout streams should be assigned to upon execution of your program. If you place a " " followed by a file or device name on the command line, stdin will be opened to that file or device. To redirect the stdout stream, place a " " followed by the file or device on the command line. This redirection of stdin and stdout does not affect whether or not file descriptors 1 and 2 are marked permanently open. If they are marked permanently open, then redirecting stdin or stdout will cause that stream to use some other file descriptor.

There are four libraries supplied with the C Language System. These are MTULIB.L, MTUFLT.L, MTUGRAPH.L, and MTUMISC.L. The MTULIB.L and MTUFLT.L are described in the AZTEC C Manual supplied by MANX Software Systems. Below are descriptions of the routines contained in the MTUGRAPH.L and MTUMISC.L libraries.

MTUGRAPH

This library contains a number of graphics and text cursor related functions. For the graphics commands, all coordinates are specified in actual screen coordinates which range from 0 to 479 for X, and 0 to 255 for Y. Source code for these routines are contained in the archive file, GRAPHSRC.L. You may use the ARCH utility program to extract any of these routines you wish to modify. This library is patterned after the MTU BASIC IGL library. You may wish to consult the IGL manual for introductory information.

```
sclear()
```

The sclear function clears the entire screen including the function key legends. The return value is undefined.

```
smove(x,y)
int x,y;
```

The smove function moves the graphics cursor to the specified X,Y location. If the specified coordinates are within the range 0 - 479 for X, and 0 - 255 for Y, then the function returns a value of 1. If the coordinates are outside that range, 0 is returned.

```
sdraw(x,y)
int x,y;
```

The sdraw function draws a line from the current graphics cursor location to the X,Y location specified. If the specified coordinates are within the range 0 - 479 for X, and 0 - 255 for Y, then the function returns a value of 1. If the coordinates are outside that range, 0 is returned.

```
srmove(rx,ry)
int rx,ry;
```

The srmove function moves the graphics cursor to the X,Y location specified by the relative coordinates, rx and ry. The new graphics cursor location is obtained by adding the coordinates of the current cursor location to the relative coordinates specified. If the resulting coordinates are within the range 0 - 479 for X, and 0 - 255 for Y, then the function returns a value of 1. If the coordinates are outside that range, 0 is returned.

```
srdraw(rx,ry)
int rx,ry;
```

The srdraw function draws a line from the current graphics cursor location to the X,Y location specified by the relative coordinates, rx and ry. The end point of the line is obtained by adding the coordinates of the current cursor location to the relative coordinates specified. If the resulting coordinates are within the range 0 - 479 for X, and 0 - 255 for Y, then the function returns a value of 1. If the coordinates are outside that range, 0 is returned.

```
penmode(mode)
int mode;
```

The penmode function is used to set the drawing mode used by various other drawing functions. Mode 0 = move. Mode 1 = draw. Mode 2 = erase. Mode 3 = flip. Add 4 to mode to enable dashed lines. If the mode specified is a legal value (i.e. 0 to 7) then the penmode function returns the previous setting of the drawing mode. If the mode specified is outside the range 0 - 7, then -1 is returned.

```
spen(x,y)
int x,y;
```

The spen function draws a line from the current graphics cursor location to the X,Y location specified, using the current drawing mode. If the specified coordinates are within the range 0 - 479 for X, and 0 - 255 for Y, then the function returns a value of 1. If the coordinates are outside that range, 0 is returned.

```
srdraw(rx,ry)
int rx,ry;
```

The srdraw function draws a line from the current graphics cursor location to the X,Y location specified by the relative coordinates, rx and ry, using the current drawing mode. The end point of the line is obtained by adding the coordinates of the current cursor location to the relative coordinates specified. If the resulting coordinates are within the range 0 - 479 for X, and 0 - 255 for Y, then the function returns a value of 1. If the coordinates are outside that range, 0 is returned.

```
unsigned int dashpat(pattern)
unsigned int pattern;
```

The dashpat function is used to set the dash pattern used while drawing dashed lines. The pattern is stored such that the most significant bit of the passed arguments controls the first dot plotted. The dashpat function returns the previous setting of the dash pattern. As a special case, if the passed argument is zero, then the current pattern is left unchanged.

```
label(pstr)
char *pstr;
```

The label function is used to print a string of characters on the display at the current graphics cursor location. The passed argument is a pointer to a null terminated string of characters from 0 to 255 characters long. It should be remembered, however, that the characters will not wrap to the next line once the right edge of the screen is reached. Characters which would be drawn past the right edge of the display simply do not appear. The label function returns the value of the X coordinate of the graphics cursor location before the string is printed.

```
sgrin(px,py)
int *px,*py;
```

The sgrin function is used to input a set of X,Y coordinates from the user, using the GRIN cursor. When this function is executed, a full screen cross hair will appear at the current graphics cursor location. The user can maneuver this cross hair about the display using the cursor keys. The speed of movement is increased while one of the SHIFT keys is held down. When a key other than one of the cursor keys is pressed, the X,Y coordinates of the cross hair position will be stored in variables pointed to by the passed arguments. The ASCII value of this key is the value returned by the sgrin function. The sgrin function does not modify the current graphics cursor location.

```
sltpen(px,py)
int *px,*py;
```

The sltpen function is used to input a set of X,Y coordinates from the user, using the light pen. When this function is executed, an attempt will be made to detect a light pen "hit" during the next full refresh of the display screen. If a light pen hit is detected, the X,Y coordinate of the location of the hit will be stored in the variables pointed to by the passed arguments. Also, the sltpen function will return a value of 1 to indicate a "hit" occured. If a light pen hit did not occur, the sltpen function returns a value of 0, and the variables are left unmodified.

```
tcursor(c,r)
int c,r;
```

The tcursor function is used to set the location of the text cursor within the current text window. If the column argument is greater than 80, or the row argument is greater than the number of lines in the text window, then the tcursor function returns a value of 0. Otherwise it returns a value of 1.

```
twindow(top,nl)
int top,nl;
```

The twindow function is used to set the text window. The "top" argument specifies the number of scan lines from the top of the display to the top of the window. This corresponds to a Y coordinate of 255 - top. The "nl" argument specifies the number of lines in the text window. If top is greater than 245 or nl greater than 25 then the arguments are ignored and the old window remains in effect. In this case, the function returns a value of 0. If the arguments are within range, the new window will be set and the function will return a value of 1.

```
sqdot(x,y)
int x,y;
```

The sqdot function is used to test the state of a particular pixel on the display. The location of the pixel is specified by the X,Y coordinates supplied as arguments. The sqdot function will return a value of 0 if the pixel is off, or a value of 1 if the pixel is on.

```
sfill(xmin,xmax,ymin,ymax)
int xmin,xmax,ymin,ymax;
```

The sfill function is used to fill a rectangular area of the display by drawing horizontal lines using the current graphics mode. The lines are drawn from xmin to xmax, starting at ymin. The horizontal line is draw repeatedly while stepping one pixel at a time vertically until ymax is reached. Some interesting patterns can be created by using dashed lines to fill the area.

```
getxy(px,py)
int *px,*py;
```

The getxy function is used to obtain the X,Y coordinates of the current graphics cursor location. The coordinates of the graphics cursor will be stored at the memory locations pointed to by the passed arguments. The getxy function itself always returns a value of 1.

```
getcur(pcol,prow)
int *pcol,*prow;
```

The getcur function is used to obtain a copy of the column and row position of the text cursor. The current column and row will be stored at the memory locations pointed to by the passed arguments. The getcur function itself always returns a value of 1.

MTUMISC.L

The MTUMISC.L library contains some miscellaneous function which may prove handy when implementing your own applications. Source code for these functions is stored in the archive file, MISCSRC.L. You may use the ARCH utility program to extract any of the routines you wish to modify.

```
legend(n,pstring)
int n;
char *pstr;
```

The legend function is used to set and redraw the legend boxes on the bottom of the display. The function will set one or more legends starting with the legend specified by the first argument. The second argument points to a null terminated string containing the legend(s) you wish to set. To set the legends properly, each legend contained in the string should occupy 8 characters. Once the legends have been stored, the legend function redraws the legends on the display. The legend function does not return a value.

```
setlgnd(n,pstr)
int n;
char *pstr;
```

The setlgnd function is identical to the legend function except that it does not redraw the legends on the display. This allows you to set the legends in several steps before displaying them.

```
getkey()
no arguments
```

The getkey function is used to get a key directly from the keyboard. The function will wait for a key to be pressed, and return the value of that key. It is most useful for inputting function key selections since the text cursor does not flash while it is waiting.

```
system(pstr)
char *pstr;
```

The system command is used to pass a command to CODOS for execution. This function will use CODOS's standard input buffer to pass the command so if an error occurs, the command will be displayed properly in the error message.

```
inline(pbuf)
char *pbuf;
```

The inline function is a specialized input line routine, optimized for use in application programs. It offers a subset of the normal CODOS line editing features, but will not allow the user to cursor out of, or past the end of the current line. When executed, the inline function also establishes the current column as the beginning point of the line and will not allow the user to cursor or delete back past this point. This prevents the user from altering a formatted screen by causing it to scroll, or cursoring on top of other data or prompts. The line editing functions supported are BS, RUBOUT, CNTL-X, LEFT-ARW, RIGHT-ARW, Shift/LEFT-ARW, Shift/RIGHT-ARW, INSERT, and DELETE. Also, the inline function is able accept an abort function key, which will automatically perform a CNTL-X and RETURN. Which function key is specified by using the setcanky function, and initially is not set to any key.

The argument specified is a pointer to the buffer into which the user's input is to be stored. This buffer should be of sufficient size to hold the maximum length of input that could be entered. Upon return, this buffer will contain the users input, terminated will a null character. The carriage return is not stored with the input. The value returned by the function will be the number of characters in the input, not counting the null character.

```
edline(pbuf,index)
char *pbuf;
int index;
```

The edline function is identical to the inline function with two exceptions. First, it assumes there is a null terminated string already in the input buffer pointed to by the first argument. This string will be displayed starting at the current text cursor position. Second, after displaying the line, the text cursor will be positioned after the one selected by the index argument. With index equal to zero, the cursor would be placed on the first character of the line. If the index is beyond the end of the string in the input buffer, the cursor is placed after the end of the string. At this point, edline function operates identically as the inline function. Refer to the inline function for details.

```
setcanky(key)
int key;
```

The setcanky function is used to set which key will automatically abort the line input, returning a null line. The key argument may be the code for any key whose bit 7 is set, but will most often be one of the function keys. Specifying a key code which is less than 128 (i.e. bit 7 not set) will effectively disable this abort feature. Refer also to the inline function.

KNOWN BUGS

As is the case with all complex pieces of software, there are always bound to be bugs present. Hopefully such bugs are either of minor importance, or happen only in rare occasions that they do not seriously affect the usefulness of the software. There are a number of known bugs in this initial release of the MTU-C Language System which are described below.

1. The Compiler (CCI) doesn't handle floating point constants correctly in assignment statements and expressions which only contain variables of type float. The solution is to make sure that one of the variables in the assignment statement or expression is of type double. For example, the following situations do not work:

```
    float a;        or        float a,b;             or        float a,b;
    a = 100;                  a = (double)100;                 a = b + 100;
```

However the following situations will work:

```
    double a;       or        float a; double b;  or         float a,b;
    a = 100;                  a = b + 200;                   a = b + (double)100;
```

2. The -X option in the MKLIB utility doesn't work properly when you specify a list of modules to extract. If you specify no list, MKLIB will correctly extract all modules in the library. These extracted modules will be slightly different from the original and as a result, it can't be recombined back into a library. You must make the libraries from originals only.

3. The 6502 Assembler (AS65.C) does not process the SVC opcode correctly. The BRK instruction is always followed by a 0 byte regardless of the argument following the SVC opcode. You should use the FCB pseudo-op to generate the proper bytes directly. For example:

```
    SVC  13
```

should be written:

```
    FCB  0,13
```

NOTES AND HINTS

This section contains some notes and hints concerning the MTU C Language System which may not be obvious from information provided elsewhere, or not mentioned the AZTEC C Manual.

1. Bit fields are not implemented in this release of the MTU C Language System.

2. The scanf() function is not implemented in this release of the MTU C Language System.

3. The creat() and open() functions in the Standard I/O make their own copy of the file or device name, converting all lower case letters to upper case. Thus, you may use upper or lower case file or device names in your programs.

4. Command line arguments may be accessed as described in section 5.11 of the Kernighan and Ritchie manual. Any arguments which redirect stdin and stdout are not included in those passed to your program through argc and argv.

5. The floating point routines use a byte alligned mantissa. This is intended to speed up calculations, but can cause some loss of precision when results are normalized. Single precision variables will always retain 17 bits of precision, which is about 5 decimal digits. Double precision variables will retain 49 bits of precision, or approximately 15 decimal digits. Floating point math will typically be slower than in other programming systems because all calculations are done in double precision.

AZTEC C for the MTU-130

USER MANUAL

RELEASE 1.00

9/6/82

Distributed by:
Manx Software Systems
201-780-4004

INTRODUCTION

The AZTEC C system consists of a comprehensive set of tools for producing systems of applications software using the C programming language. The basic system consists of a compiler, relocating assembler, and linkage editor. The AZTEC C software system runs on any MTU-130 with at least one disk drive. There are no special terminal requirements.

The major components of the AZTEC C software development system are the AZTEC C pseudo-code compiler, the AZTEC relocating pseudo-code assembler, the AZTEC relocating 6502 assembler, and the AZTEC linkage editor.

The standard reference for the C language is:

> Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language. Prentice-Hall Inc., 1978, (Englewood Cliffs, N.J.)

Dennis Ritchie originally designed C for the UNIX project. The above text besides providing the standard definition and reference for the C language is an excellent tutorial. AZTEC C can be conveniently used in conjunction with the K & R text for learning the C language.

The AZTEC C compiler for the MTU-130 is a pseudo-code compiler. The pseudo-code compiler produces code that is assembled by the AZTEC pseudo-code assembler to produce relocatable object files that are interpreted at run time. Pseudo-code and native code object files are freely mixed by the AZTEC linker to produce hybrid modules that will run both native and interpreted. AZTEC C is written almost entirely in C and is implemented using the pseudo-code compiler.

AZTEC C supports all of the C language features except for bit fields. It is suitable for writing system, utility, or text processing software. The AZTEC C system consists of a compiler that produces pseudo-code assembler source, a relocating assembler, and a linkage editor plus a comprehensive set of run time library routines.

The AZTEC ASI relocating assembler produces relocatable object files that are combined with other relocatable object files and library routines by the AZTEC LN linkage editor. The linkage editor will scan through one or more run time libraries and incorporate any routines that are referenced by the linked modules.

TRADEMARKS

AZTEC CCI, AZTEC ASI, AZTEC AS65, and AZTEC LN are trademarks of Manx Software Systems. UNIX is a trademark of Bell Laboratories.

AZTEC C Compiler System - General Overview

The AZTEC C Compiler System for the MTU-130 is not just a single program. It is a set of programs which transform the MTU-130 computer into a complete program development system.

The C compiler itself consists of four separate programs, the C pseudo-code compiler, CCI, the two relocating assemblers, ASI and AS65, and the linker, LN.

The system is supplied on one single-sided floppy diskette labelled optional. software diskette. The disk contains the following files:

CCI.C	ASI.C	AS65.C
LN.C	MKLIB.C	ARCH.C
CMP.C	SIZE.C	OD.C
NM.C		
STDIO.H	MTULIB.L	MTUFLT.L
	STDIOSRC.L	
PROGSRC.L		

C Compiler Usage - General Overview

This section gives a general overview of the process of creating a runnable binary image from a C language source file. More detailed information on each of the programs involved in the various steps is available in the following sections.

The first step is to create a C language source file. This can be done using the visual editor, EDIT, supplied with the system, or by using some other editor. For example, using EDIT, type the following simple program and call it "HELLO.T":

```
main()
{
        printf("Hello world!!\n");
{
```

The second step is to compile the program into the pseudo-code assembly language. This is accomplished using the CCI program. Type:

```
      CCI -O HELLO.I HELLO.T
```

The compiler will load and display a startup message giving the copyright message and the version number. The compiler will then read the C source file and place the compiled result into the file "HELLO.I". If there are any error messages, they will be displayed on the screen by showing the line where an error was detected and the error number itself. Correct any errors using the editor and compile the program again. Continue this procedure until the compiler runs without producing any diagnostic messages.

The third step is to produce a relocatable object module from the pseudo-code assembly language using the AZTEC Pseudo-code Assembler, ASI. Type:

```
      ASI -O HELLO.R HELLO.I
```

This program also displays a startup message when executed. This program should run to completion without any problems.

The object module must now be combined with the library routines and the result placed into a binary file which can be executed. To do this, type:

```
      LN -O HELLO.C HELLO.R MTULIB.L
```

LN is the C Linker and is used to combine one or more object modules with whatever library routines are needed to produce an executable binary image. MTULIB.L is the standard library archive.

When LN finishes, the process is complete. To run the program, simply type "HELLO" to the CODOS prompt followed by a return. The program will be loaded from the disk and the message:

```
      Hello world!!
```

should appear.

AZTEC C PSEUDO-CODE COMPILER

The AZTEC C pseudo-code compiler is implemented according to the language description supplied by Brian W. Kernighan and Dennis M. Ritchie, in The C Programming Language. The user should refer to that document for a description and definition of the C language. This document will detail areas where the AZTEC C compiler differs from the description in that book.

The reader who is not familiar with C and does not have a copy of the Kernighan and Ritchie book is strongly advised to acquire one. The book provides an excellent tutorial for learning and using C. The program examples given in the book, can be entered, compiled with AZTEC C and executed to reenforce the instruction given in the text.

The library routines defined in standard C that are supported by AZTEC C are identical in syntax to the standard. The library routines that are supported are defined in the library section of this manual. AZTEC C includes some extended library routines, that do not exist in the standard C, to allow access to native operating system functions. These are also described in the library section. The system dependent functions should be avoided in favor of the standard functions if there is or may be a requirement to run the software under different operating systems.

AZTEC C is invoked by the command:

```
CCI NAME.T
```

It is recommended that the file name end in ".T", but it is not necessary. C source statements found in the "NAME.T" file are translated to pseudo-code assembler source statements and written to a file named "NAME.I". If some other name is wanted, then the "-O" option is used. For example:

```
CCI -O TEMP.I DBMS.T
```

will process the C statements in "DBMS.T" and write the translated source to "TEMP.I".

By default, AZTEC C expects that pointer references to members within a structure are limited to the structure associated with the pointer. However, to support existing source where this is not the case, the "-S" option is provided. If the "-S" is specified as a compile time option and a pointer reference is to a member name that is not defined in the structure associated with the pointer then all previously defined structures will be searched until the specified member is found. The search will begin with the structure most recently defined and search backward from there.

The "-T" option will copy the C source statements as comments in the assembly language output file. Each C statement is followed by the pseudo-code generated from that statement.

There are four options for changing default internal table sizes. The "-E" option specifies the size of the expression work table. The "-X" option specifies the size of the macro (#define) work table. The "-Y" option specifies the maximum number of outstanding cases allowed in a switch statement. The "-Z" option specifies the size of the string literal table.

The default value for "-E" is 120 entries. Each entry uses 14 bytes. Each operand and operator in an expression requires one entry in the expression table. Each function and each comma within an argument list is an operator. There are some other rules for determining the number of entries that an expression will require. Since they are not straightforward and are subject to change, they will not be defined here. The best advice is that if a compile terminates because of an expression table overflow (error 36), recompile with a larger value for "-E".

The following expression uses 15 entries in the expression table:

```
a = b + function(a + 7, b, d) * x;
```

The following will reserve space for 300 entries in the expression table:

```
CCI -E300 PROG.T
```

There must be no space between the "-E" and the entry size.

The macro table size defaults to 2000 bytes. Each "#define" uses four bytes plus the total number of bytes in the two strings. The following macro uses 9 bytes of table space:

```
#define     v     0x1f
```

The following will reserve 4000 bytes for the macro table:

```
CCI -X4000 PROG.T
```

The macro table needs to be expanded if an error 59 (macro table exhausted) is encountered.

The default size for the case table is 200 entries, with each entry using 4 bytes.

The following will use 4 (not 5) entries in the case table:

```
        switch (a)   {
        case 0:
                a += 1;
                break;
        case 1:
                switch(x)   {
                case 'a':
                        funct1(a);
                        break;
                case 'b':
                        funct2(b);
                        break;
                {
                a = 5;
        case 3:
                funct2(a);
                break;

        {
```

The following allows for 300 outstanding case statements:

```
        CCI -Y300 PROG.T
```

The size of the case table needs to be increased if an error 76 (case table exhausted) is encountered.

The size of the string table defaults to 2000. Each string literal occupies a number of bytes equal to the number of characters in the string plus one (for the null terminator).

The following will reserve 3000 bytes for the string table:

```
        CCI -Z3000 PROG.T
```

The size of the string table needs to be increased if an error number 2 (string space exhausted) is encountered.

The name of the C source file must always be the last argument.

PSEUDO-CODE ASSEMBLER

The AZTEC ASI assembler is a relocating assembler and is invoked by the command line:

```
ASI NAME.I
```

The relocatable object file produced by the assembly will be named "NAME.R" where name is the same name as the name prefix on the ".I" file. An alternative object filename can be supplied by specifying "-O NAME.R". The object file will be written to the filename following "-O". The filename does not have to end with ".R", it is, however, the recommended format. The file "NAME.I" is the pseudo-code assembly language source file. The filename does not have to end in ".I".

A complete written definition of the pseudo-code and the syntax are not currently available.

6502 ASSEMBLER

A. Overview

The AZTEC AS65 assembler is a relocating assembler which supports most of the standard MOS Technology mnemonics and is invoked by the command line:

```
AS65 TEST.A
```

The relocatable object file produced by the assembler will be named "TEST.R" where test is the same name as the prefix of the input filename. An alternative object filename can be supplied by specifying the "-O" option, as in "-O NAME.R". The object file will be written to the filename following the "-O". The filename does not have to end in ".R", it is, however, the recommended format. The file "TEST.A" is the assembly language source file. The filename does not have to end in ".A".

There is no way currently to produce an assembly listing. A future version may provide that option.

The following defines the syntax for the AS65 assembler.

B. Statements

Source files for the AZTEC AS65 assembler consist of statements of the form:

```
[label]     [opcode]    [argument] [[;]comment]
```

The brackets "[...]" indicate an optional element.

C. Labels

A label consists of any number of alphanumerics starting in column one. If a statement is not labeled, then column one must be a blank or a tab or an asterisk. An asterisk denotes a comment line. A label must start with an alphabetic. An alphabetic is defined to be any letter or one of the special characters '_' or '.'. An alphanumeric is an alphabetic or a digit from 0 to 9. A label followed by "#" is declared external. The AZTEC C compiler places a '_' character at the end of all labels that it generates.

D. Expressions

Expressions are evaluated from left to right with no precedence as to operator or parentheses. Operators are:

* -	multiply
/ -	divide
+ -	add
- -	subtract
# -	constant
= -	constant
< -	low byte of expression
> -	high byte of expression

E. Constants

The default base for numeric constants is decimal. Other bases are specified by the following prefixes or suffixes:

BASE	PREFIX	SUFFIX
2	%	b,B
8	@	o,O,q,Q
10	null,&	null
16	$	h,H

A character constant is of the form 'character as in 'A.

F. Assembler Directives

The AZTEC AS65 assembler supports the following pseudo operations:

COMMON block name	sets the location to the selected common block.
CSEG	select code segment.
DSEG	select data segment.
END	end of assembler source statements.
ENTRY expr	entry point of final module.
EQU expr	define label value.
FCB expr	define byte constant.
FCC /expr/	define byte string constant.
FDB expr	define double byte constant.

FUNC label	if label is not defined then it is declared external.
INSTXT file	the specified file is included at this point.
PUBLIC label	declares label to be external.
RMB expr	reserves expr bytes of memory with no particular value.
WEAK expr	define label value if not previously defined.

G. Interfacing Assembly Language Routines with C Programs

The calling conventions used by the AZTEC C compiler are very simple. The arguments to a function are pushed onto the stack in reverse order, i.e. the first argument is pushed last and the last argument is pushed first. Since the 6502 has only a 256 byte hardware stack, a pseudo-stack is implemented using zero-page locations 2 and 3.

The function is called using the 6502 JSR instruction and the return address is located on the hardware stack. The first argument is located at the address contained in locations 2 and 3. When the function returns, the arguments are removed from the stack. A function is required to return with the arguments still on the stack unless something is pushed back in place of them.

Locations 2 through 31 are reserved for use by the pseudo-code interpreter as defined in the following table:

Locations	Use
2- 3	Pseudo stack pointer.
4- 5	Pseudo frame pointer.
6- 7	Pseudo program counter.
8-11	Pseudo register R0.
12-15	Pseudo register R1.
16-19	Pseudo register R2.
20-21	Pointer to floating point accumulator.
22-23	Pointer to floating point secondary.
24-31	Miscellaneous.

The function's return value should be in R0.

H. Examples

```
*
*  Copyright (c) 1982 by Jim Goodnow II
*
VAL     EQU     0
SP      EQU     2
FRAME   EQU     4
PC      EQU     6
R0      EQU     8
R1      EQU     12
R2      EQU     16
*
*       isupper(c);
*
        public  isupper_
isupper_
        ldy     #0          ;offset of zero
        lda     (SP),Y      ;get the character off the stack
        cmp     #'A         ;compare to character 'A'
        bcc     false       ;if less, goto false
        cmp     #'Z+1       ;compare to character 'Z'+1
        bcs     false       ;if greater or equal, goto false
*
true    sty     R0+1        ;set high byte to 0
        iny                 ;get a one
        sty     R0          ;set low byte to 1
        rts
*
*       islower(c);
*
        public  islower_
islower_
        ldy     #0          ;offset of zero
        lda     (SP),Y      ;get the character off the stack
        cmp     #'a         ;compare to the character 'a'
        bcc     false       ;if less, goto false
        cmp     #'z+1       ;compare to the character 'z'+1
        bcc     true        ;if less, goto true
*
false   sty     R0          ;set low byte to 0
        sty     R0+1        ;set high byte to 0
        rts
```

LINKER

The AZTEC LN link editor will combine object files produced by the AZTEC ASI pseudo-code assembler and/or by the AZTEC AS65 6502 assembler, select routines from object libraries created with the MKLIB or ARCH utility and produce an executable binary file.

Supplied with the MTU C Compiler System is the MTULIB.L object library. In almost all cases this library must be specified. To link a simple single module routine, the following command will suffice:

```
LN NAME.I MTUDOS.L
```

The operand "NAME.I" is the name of the object file. The executable file created by LN will be named "NAME.C". The "-O" option followed by a filename can be used to create an alternative name for the LN output file.

Several modules can be linked together as in the following example:

```
LN -O NAME MOD1.I MOD2.I MOD3.I MTULIB.L
```

Also several libraries can be searched as in the following:

```
LN -O NAME MOD1.I MOD2.I MINE.L MINE.L MTULIB.L
```

Libraries are searched sequentially in order of specification. It is expected that all external references are forward. One way to deal with the problem of routines that make external reference to a routine already passed by the librarian is the following:

```
LN -O NAME MOD1.I MINE.L MINE.L MTULIB.L
```

The link editor will read the "MINE.L" library twice. The second time through it will resolve backward references encountered on the first pass.

Other options for the link editor include:

```
-T
```

to create a symbol table for debugging purposes. The symbol table file will have the same prefix name as the output file with a suffix of ".S".

```
-B address
```

to specify a base address other than hex 700. The "base address" is assumed to be in hex.

-C address

to specify a starting address for code portion of the output. The default is the base address + 3. The first three bytes are usually occupied by a jump instruction to system initialization code. It is assumed that the code starting address is specified as a hex number.

-D address

to specify a data address. Data is usually placed behind the end of the code segment.

-F filename

to merge the contents of "filename" with command line arguments. More than one specification of "-F" can be supplied. There are several advantageous uses for this command. The most obvious is to supply the names of modules that are commonly linked together. All records in the file are read. There is no need to squeeze everything into one record.

-S address

to specify the initial value of the pseudo-stack used by the program. The default value is hex BDF0.

Utility Programs

Along with the compiler, assemblers and linker, there are a number of utility programs provided on the disk. These programs are fairly simple in design, but are useful both as tools and as examples to study. The source to several of these utilities is included on the disk in the archive "PROGSRC.L".

A. OD

```
OD file1 [file2] [file3] ...
```

This program performs a binary dump in hex and ascii of the specified file to the standard output. The program continues until the end of the file and then dumps the next file if any.

B. CMP

```
CMP [-L] file1 file2
```

This program compares two files on a character by character basis. When it finds a difference, a message is printed giving the offset from the beginning of the file. If the '-L' option is given, the program will list all differences otherwise it will stop after the first difference. If no difference is found, a message will be printed to that effect.

C. SIZE

```
SIZE file1.R [file2.R] ...
```

This program operates only on the relocatable object files which are the output of the two assemblers, ASI and AS65. It examines the header of each file specified and prints the size of the text, initialized data, uninitialized data and the total of all three. The total is printed in both decimal and hex.

D. NM

```
NM [-UNAGO] file1.R [file2.R] ...
```

This utility also operates only on relocatable object files. This program prints the symbol table (name list) of each object file. The output consists of a symbol name preceded by the value of that symbol. Between the symbol name and its value is a character indicating the type of symbol. The characters used are:

A - absolute
T - program text
D - initialized data
C - common
R - reference to common
E - expression
U - undefined
W - weak definition

The options available are:

-g	Print only global (external) symbols.
-u	Print only undefined symbols.
-n	Sort numerically.
-a	Sort alphabetically.
-o	When multiple file names are given, each name is printed before the name list for that file. When this option is given, the file name is printed at the beginning of each line.

E. ARCH

```
ARCH -[CLVDXA]O archive [file1] [file2] ...
```

This program is used to create and manipulate archive files. Archive files are used as a convenient means of collecting source modules together and as a library to be searched by the linker. The 'O' option must always be used to specify the name of the archive itself. Only one of the options 'LDXA' may be specified. The 'V' option is a modifier for the 'DXA' options and causes them to print each file name they act upon. The following table gives the meaning of each option.

L	list the named files in the archive, giving name and size of each. If no file names are specified, all files are listed.
A	add the specified files to the end of the archive. If the 'C' option is given as well, the archive is truncated before adding the files.
X	extract the named files from the archive. If no file names are given, all files are extracted from the archive. The archive is not modified.
D	delete the named files from the archive.

Different types of files may be freely intermixed within an archive file.

F. MKLIB

```
MKLIB [-ALX] -O LIBRARY MOD1 MOD2 ...
```

This program creates libraries in the most efficient form. Each module is rearranged to make the linking process as fast as possible. In addition, a dictionary of global variables which are resolved in the modules is automatically created. This dictionary is used by the linker so that it only looks at those modules that it needs to. The following table gives the meaning of each option.

A	add the specified modules to the end of the library.
L	list the names of the modules in the library.
X	extract the named modules from the library.

Libraries

A. Overview

The libraries provided with the MTU C system can be divided into four logical groupings. These groups are the standard I/O, system I/O, floating point and utility libraries.

The compiled object modules are supplied as two archives on the disk named MTULIB.L and MTUFLT.L. MTULIB.L contains the standard I/O, the utility routines and the system I/O routines. MTUFLT.L contains all routines having anything to do with floating point.

MTUFLT.L and MTULIB.L should be the last two arguments to the linkage loader. MTUFLT.L need only be specified if floating point is used in some routine. In that case, MTUFLT.L should appear before MTULIB.L.

The following sections describe each library routine's calling sequence and function. Some of the routines are written in 6502 assembly language to improve performance.

B. Standard I/O

The standard I/O library is based on the set of routines developed for use on UNIX operating systems. Use of standard I/O guarantees compatibility at the source level with both UNIX and other Aztec C systems. Standard I/O is actually a set of routines which use the system I/O routines to perform the actual input and output. Standard I/O is fairly efficient when doing character at a time I/O, since the data is buffered and there is no need to call CODOS for each byte.

The buffer size used defaults to 256. The buffer space is not allocated until the first attempt to get or put a byte. Thus, the size of the buffer can be changed between the call to fopen and the first use of the stream as follows:

```
FILE *fp;

fp = fopen("FILE.T", "r");
fp->bufsiz = 2048;
```

The buffer is allocated using the malloc() routine, and released when the file is closed by using the free() routine.

To use the standard I/O package, you should insert the statement:

```
        #include   "STDIO.H"
```

into your programs to define the FILE data type and miscellaneous other things needed to use the functions.

There is one significant difference between the implementation of standard I/O on UNIX and in the MTU C System. In the UNIX operating system, the end of line delimiter is the newline character which is normally specified as a backslash followed by the letter n, as in '\n'. This corresponds to a hex A, or line feed character. The MTU system, on the other hand, normally uses a hex D, or carriage return to end a line. This presents a problem.

The solution that has been adopted is to provide "raw" getc and putc routines as well as ASCII versions called agetc and aputc. The ASCII versions translate newlines to carriage returns on output, and carriage returns to newlines on input. The following routines do no translation:

getc	putc	fgetc	fputc	fread	fwrite
getw	putw	ungetc			

The following routines do the translation:

agetc	aputc	gets	fgets	printf	fprintf
sprintf	puts	fputs	getchar	putchar	

In the following section, each routine will be described by specifying the routine name, the arguments and their types, the language in which the routine is written, the action performed by the routine and the return values if any.

B.1. clearerr(stream) **C**
FILE *stream;

Clearerr resets the error indication on the named stream.

B.2. exit(n) **C**
int n;

Flushes and closes all open steams. Returns to the operating system with n as the return code. N is not currently used, but may be used by future versions.

B.3. fclose(stream) **C**
FILE *stream;

Flushes any data not yet written out and closes the named stream. EOF is returned if any errors occur, otherwise 0.

B.4. feof(stream) **C Macro**
FILE *stream;

Returns non-zero if end of file has been encountered on the named stream.

B.5. ferror(stream) **C Macro**
FILE *stream;

Returns non-zero if an error has occurred reading or writing on the named stream. The error indication will remain with the stream until cleared using clearerr or until the stream is closed.

B.6. fflush(stream) **C Macro**
FILE *stream;

If the steam has been opened for write, any data which has been buffered, is written out.
EOF is returned if any errors occur, otherwise 0.
(The real work is done by _flsh.)

B.7. fgetc(stream) **6502 Assembly**
FILE *stream;

Returns the next character from the named stream. EOF is returned on end of file and is distinguishable from the regular data.

B.8. char *fgets(buf, limit, stream) **C**
char *buf;
int limit;
FILE *stream;

Reads n-1 characters or up to a newline character from the named stream. The last character read is followed by a null character. Fgets returns its first argument.

B.9. fileno(stream) **C Macro**
FILE *stream;

Returns the file descriptor used by the system I/O routines associated with the named stream. Note that intermixing standard I/O calls and system I/O calls on the same file may not have the desired effect because of the buffering done by standard I/O.

B.10. FILE *fopen(name, mode) **C**
char *name, *mode;

Opens a file or device according to "mode" and returns a pointer to an open I/O stream which can be used with other standard I/O functions. Name is a pointer to a string containing the name of the device or file in the same form as used by the SHELL.

NULL is returned if there is an error opening the file. The valid values for "mode" are:

"r"	The file is opened for reading only.
"r+"	The file is opened for reading and writing.
"w"	The file is opened for writing. If "name" exists, it is deleted and a new one is created.
"w+"	Same as "w" except the file is opened for reading and writing.
"a"	The file is opened for writing. The file is positioned at the end of file.
"a+"	Same as "a" except the file is opened for reading and writing.

B.11. fprintf(stream, format, arg1, arg2, ...) C

```
FILE *stream;
char *format;
```

Formats data according to format and writes the result to the named stream. Formatting is done as described in Chapter 7, Input and Output, of The C Programming Language.

B.12. fputc(c, stream) 6502 Assembly

```
int c;
FILE *stream;
```

Appends the character c to the named stream. It returns EOF on error, otherwise it returns the character written.

B.13. fputs(str, stream) C

```
char *str;
FILE *stream;
```

Writes the null-terminated string pointed at by str to the named stream.
EOF is returned on error.

B.14. fread(buf, size, nitems, stream) C

```
char *buf;
unsigned size;
int nitems;
FILE *stream;
```

Reads 'nitems' of length 'size' at the address pointed to by 'buf' from the named stream. Returns the number of completely read items.

B.15. **FILE *freopen(name, mode, stream)** **C**
char *name, *mode;
FILE *stream;

Substitutes the named file in place of the open stream after closing it. It returns the stream. This is normally used to attach the preopened constant names, stdin, stdout, and stderr to specified files.

B.16. **fseek(stream, offset, how)** **C**
FILE *stream;
long offset;
int how;

Positions the named stream according to how and offset. If how is 0, file is positioned offset bytes from the beginning of the file. If how is 1, file is positioned offset bytes relative to the current position in the named stream. If how is 2, file is positioned offset bytes relative to the end of file of the named stream. Fseek returns EOF if an error occurs.

B.17. **long ftell(stream)** **C**
FILE *stream;

Returns the current position in the named stream.

B.18. **fwrite(buf, size, nitems, stream)** **C**
char *buf;
unsigned size;
int nitems;
FILE *stream;

Writes 'nitems' of length 'size' from the address pointed to by 'buf' onto the named stream. Returns the number of completely written items.

B.19. **getc(stream)** **6502 Assembly**
FILE *stream;

Returns the next character from the named stream. EOF is returned on end of file and is distinguishable from the regular data.

B.20. **getchar()** **C, C Macro**

Returns the next character from the standard input, as in agetc(stdin).
This routine is defined as both a macro and a routine, for the common case where "stdio.h" is not included.

B.21. char *gets(buf) **C**
char *buf;

Reads characters from the standard input until end of file or a newline character are received. The newline, if received, is placed in the buffer. The last character received is followed by a null character. Gets returns its argument unless an end of file is received before any other characters are received, in which case it returns 0.

B.22. getw(stream) **C**
FILE *stream;

Returns the next word (two bytes) from the named stream. The constant EOF is returned on end of file or an error. However, since EOF is a valid integer, feof and ferror must be used to determine the success of the call.

B.23. printf(format, arg1, arg2, ...) **C**
char *format;

Formats data according to format and writes the result to the standard output stream, stdout. Formatting is done as described in Chapter 7, Input and Output, of The C Programming Language.

B.24. putc(c, stream) **6502 Assembly**
char c;
FILE *stream;

Appends the character c to the named stream. It returns EOF on error, otherwise it returns the character written.

B.25. putchar(c) **C, C Macro**
char c;

Writes the character c to the standard output, as in putc(c, stdout). This routine is defined as both a macro and a routine, for the common case where "stdio.h" is not included.

B.26. puts(str) **C**
char *str;

Writes the null-terminated string pointed at by str to the standard output. Unlike fputs, puts appends a newline to the string. EOF is returned on any errors.

B.27. putw(w, stream) **C**
int w;
FILE *stream;

Writes the word w to the named stream. Putw returns the word written. If an error occurs, it returns EOF which is a good integer, so ferror shoud be used to detect errors.

B.28. rewind(stream) **C**
FILE *stream;

Positions the named stream at its beginning, as in fseek(stream, 0L, 0).

B.29. setbuf(stream, buf) **C**
FILE *stream;
char *buf;

Setbuf is used to specify that the buffer buf be used instead of a system allocated one. Setbuf is used on an open stream before it has been read or written. If buf is the constant NULL, I/O will be unbuffered.

B.30. sprintf(str, format, arg1, arg2, ...) **C**
char *str, *format;

Formats data according to format and stores the null-terminated result in the buffer pointed at by str. Formatting is done as described in Chapter 7, Input and Output, of <u>The C Programming Language</u>.

B.31. ungetc(c, stream) **C**
char c;
FILE *stream;

Pushes the character c back on the named stream. The character will be returned by the next getc call on that stream. Normally returns c and returns EOF if c cannot be pushed back. Only one character of pushback is guaranteed and EOF cannot be pushed back.

C. System I/O

The system I/O routines are really just an interface to CODOS. A file or device is accessed by first using the "open" library routine. This routine returns an integer from 0 through 6. This value is used in subsequent "read", "write" and "close" calls and is called a file descriptor.

When a program is initiated from CODOS, the first code that is executed are the CRT0/1 startup routine and the croot() command parser. CRT0 is a 6502 assembly language routine that initializes the stack and calls croot(). CRT1 is similar to CRT0, but is loaded when floating point is being used. The C language routine croot() pre-opens the file descriptors 0, 1 and 2 which default to the keyboard and screen. This is a carry-over from the UNIX operating system convention of using file descriptor 0 for standard input, file descriptor 1 for standard output, and file descriptor 2 for the standard error output. These file descriptors may be closed and reopened to other devices or files. If the program is called using the '<' and/or '>' I/O redirection, the croot routine opens file descriptor 0 to the file name after the '<' and opens file descriptor 1 to the file name after the '>'.

The following is a description of each of the system i/o library calls.

C.1. creat(name, mode) **C**

```
char *name;
int mode;
```

Creat is used to open a file for writing which may or may not exist. If the file does not exist, the file is created. If the file does exist, the file is truncated. In either case, the file is opened for the mode indicated (see open) and the file descriptor returned. If the creat fails, the value returned is negative. This value indicates the type of error which occurred.

C.2. open(name, mode) **C**

```
char *name;
int mode;
```

Open is used to get access to the file or device 'name' and returns a file descriptor upon a successful open. The value of mode is 0 if the file is to be opened for read only, 1 if opened for write only, and 2 if opened for reading and writing. When a file is opened, it is positioned at the beginning.

If the open fails, the value returned is negative and indicates the type of error which occurred.

C.3. read(fd, buf, cnt) 6502 Assembly

```
int fd;
char *buf;
int cnt;
```

This routine attempts to read 'cnt' characters into the buffer whose address is given by 'buf' from the file or device specified by 'fd'. The actual number of characters read is returned. A zero is returned if at the end of a file and a negative value if an error occurs.

C.4. write(fd, buf, cnt) 6502 Assembly

```
int fd;
char *buf;
int cnt;
```

This routine writes 'cnt' characters from the buffer whose address is given by 'buf' to the file or device specified by 'fd'. The actual number of characters written is returned. A negative error number is returned if an error occurs.

C.5. long lseek(fd, pos, how) **C**

```
int fd;
long pos;
int how;
```

Positions the file specified by fd according to how and pos. If how is 0, the file is positioned at pos bytes from the beginning. If how is 1 or 2, the file is positioned pos bytes from the current position or end of file, respectively, where pos may be negative. In any case, the new position is returned. If an error occurs, a negative error number is returned.

C.6. close(fd) **6502 Assembly**

```
int fd;
```

The file or device specified by fd is closed.

C.7. ioctl(fd, cmd, arg) **C**

```
int fd, cmd, arg;
```

This function currently returns 0 if the device is not block oriented, 1 if it is, and -1 if the fd has not been opened. This routine is used by the standard I/O routines to determine whether the device should be buffered or not.

C.8. _exit(n) **C**

```
int n;
```

Returns control to the operating system command parser after closing all open files. The value 'n' is not used at present but may be used in future versions.

C.9. unlink(name) **C**

```
char *name;
```

The named file is deleted. Return is zero, or the negative error number if an error occurs.

C.10. rename(old, new) **C**

```
char *old, *new;
```

The file named 'old' is renamed as 'new'. Return is zero, or the negative error number if an error occurs.

D. Floating Library

The floating point library consists of three user callable subroutines and the necessary support routines which are called by the interperter to perform floating point operations. The user callable routines are defined below.

D.1. double atof(str) **6502 Assembly**
char *str;

Convert the null terminated ASCII string pointed to by str into a floating point number.

D.2. ftoa(flt, str, precision, type) **6502 Assembly**
double flt;
char *str;
int precision, type;

Convert from float/double format to an ASCII string. The value of flt is converted to ASCII and stored in the space pointed to by str. Precision specifies the number of digits to the right of the decimal point. If type is 0, the format will be of the printf F type. If type is 1, the string will be in E format. This is the routine used by format.

D.3. format(outp, str, args) **C**
int (*outp)();
char *str;
unsigned *args;

Formats data according to the string, str, and calls the given function, outp, with each character of the result. Formatting is done as described in Chapter 7, Input and Output, of _The C Programming Language_. The version of format in this library differs from the version in the standard library by supporting the %E and %F formats.

E. Utility Routines

The utility functions consist primarily of routines which deal with memory allocation and manipulation and string functions. Most of these routines are written in 6502 assembly language to improve performance.

E.1. atoi(str) **C**
char *str;

Converts a string of decimal digits into a signed integer value. Conversion stops on the first non-numeric value encountered in the string.

E.2. long atol(str) **C**
char *str;

Converts a string of decimal digits into a signed long value. Conversion stops on the first non-numeric value encountered in the string.

E.3. blockmv(dest, src, cnt) **6502 Assembly**
```
char *dest, *src;
int cnt;
```

This routine moves cnt bytes from dest to src. The routine checks for overlap and starts at the appropriate end.

E.4. char *index(str, c) **6502 Assembly**
```
char *str;
char c;
```

Returns a pointer to the first occurence of character c in the null-terminated string str. Returns 0 if c is not found.

E.5. char *rindex(str, c) **6502 Assembly**
```
char *str;
char c;
```

Returns a pointer to the last occurence of character c in the null-terminated string str. Returns 0 if c is not found.

E.6. strlen(str) **6502 Assembly**
```
char *str;
```

Returns the length of the string str (not including the terminating null character.)

E.7. char *strcat(s1, s2) **6502 Assembly**
```
char *s1, *s2;
```

Appends a copy of string s2 to the end of string s1. A pointer to the null terminated result is returned.

E.8. char *strncat(s1, s2, n) **6502 Assembly**
```
char *s1, *s2;
```

Appends at most n characters of string s2 to the end of string s1. A pointer to the null terminated result is returned.

E.9. strcmp(s1, s2) **6502 Assembly**
```
char *s1, *s2;
```

Compares its arguments character by character and returns an integer greater than, equal to, or less than 0, according as s1 is lexicographically greater than, equal to, or less than s2.

E.10. strncmp(s1, s2, n) **6502 Assembly**
```
char *s1, *s2;
```

Same comparison as strcmp, but compares only up to n characters.

E.11. strcpy(s1, s2) **6502 Assembly**
char *s1, *s2;

Copies string s2 to string s1 including the null character. Returns s1.

E.12. strncpy(s1, s2, n) **6502 Assembly**
char *s1, *s2;

Copies at most n characters of string s2 to string s1. If a null character is encountered before the n'th character, s1 will be padded with nulls. If n characters are copied and a null character has not been copied no null is placed in s1. Returns s1.

E.13. tolower(c) **C**
char c;

Returns the character c in lower case if it is a lower case alphabetic, otherwise c is returned unchanged.

E.14. toupper(c) **C**
char c;

Returns the character c in upper case if it is a lower case alphabetic, otherwise c is returned unchanged.

E.15. char *malloc(size) **C**

Returns a pointer to size bytes. This space is allocated starting at the end of the current program. A map is kept of allocated space. The companion routine free(), returns space to the usable pool. If a large enough piece of memory is not available from the poo, the top of allocated memory is remembered and each call adds the size requested to the top value. If the top value is within 200 bytes of the bottom of the stack, null is returned. Note, that the amount of space allocated is actually two bytes larger to store the size.

E.16. char *calloc(nelem, elsize) **C**
unsigned nelem, elsize;

Calls malloc asking for (nelem*elsize) bytes of memory. Included for completeness.

E.17. _setbot(mem) **C**
char *mem;

Sets the malloc routine's idea of where the first byte of usable space is. Called by each program's initialization routine.

E.18. **free(addr)** **C**
```
char *addr;
```

Deallocates a previously malloc'ed or calloc'ed piece of memory. Adjacent free spaces are collapsed.

E.19. **clear(start, len, fillc)** **6502 Assembly**
```
char *start;
int len;
char fillc;
```

Fills the len bytes starting at start with the character fillc.

ERROR MESSAGES

ERROR NUMBER	EXPLANATION
1	bad digit in octal constant
2	string space exhausted (see COMPILER -Z option)
3	unterminated string
4	compiler error in effaddr
5	illegal type for function
6	inappropriate arguments
7	bad declaration syntax
8	name not allowed here
9	must be constant
10	size must be positive integer
11	data type too complex
12	illegal pointer reference
13	unimplemented type
14	unimplemented type
15	storage class conflict
16	data type conflict
17	unsupported data type
18	data type conflict
19	too many structures
20	structure redeclaration
21	missing 's
22	struct decl syntax
23	undefined struct name
24	need right parenthesis
25	expected symbol here
26	undefined structure/union member
27	illegal type CAST
28	incompatible structures
29	structure not allowed here
30	missing : on ? expr
31	call of non-function
32	illegal pointer calculation
33	illegal type
34	undefined symbol
35	typedef not allowed here
36	no more expression space (see COMPILER -E option)
37	invalid expression
38	no auto. aggregate initialization
39	no strings in automatic
40	this shouldn't happen
41	invalid initializer
42	too many initializers
43	undefined structure initialization
44	too many structure initializers
45	bad declaration syntax
46	missing closing bracket
47	open failure on include file
48	illegal symbol name

49	allready defined
50	missing bracket
51	must be lvalue
52	symbol table overflow
53	multiply defined label
54	too many labels
55	missing quote
56	missing apostrophe
57	line too long
58	illegal # encountered.
59	macro table full (see COMPILER -X option)
60	output file error
61	reference of member of undefined structure
62	function body must be compound statement
63	undefined label
64	inappropriate arguments
65	illegal argument name
66	expected comma
67	invalid else
68	syntax error
69	missing semicolon
70	bad goto syntax
71	statement syntax
72	statement syntax
73	statement syntax
74	case value must be integer constant
75	missing colon on case
76	too many cases in switch (see COMPILER -Y option)
77	case outside of switch
78	missing colon
79	duplicate default
80	default outside of switch
81	break/continue error
82	illegal character
83	too many nested includes
84	too many dimensions
85	not an argument
86	null dimension
87	invalid character constant
88	not a structure
89	invalid storage class
90	symbol redeclared
91	illegal use of floating point type
92	illegal type conversion
93	illegal expression type for switch
94	bad argument to define
95	no argument list
96	missing arg
97	bad arg
98	not enough args
99	conversion not found in code table