Aztec C65
For the Apple][

CII Version 1.05i 6502

Apple][Shell 2.4

Portions

Mini Manual

Copyright and Conditions of Use

This manual and its related compilers have been preserved, expanded upon, and made available for the purposes of preserving the history of Aztec C65 and the Apple][on behalf of the Apple][community to provide both a virtual and real experience of that era for the enjoyment of other like-minded souls. Harry Suckow (the Copyright holder for Aztec C) has given permission to redistribute Manx Software Systems discontinued Aztec C compiler packages for now-obsolete platforms. Your use must be Fair as it applies to Manx's Copyright on these compilers.

You may use these compilers, the original routines and derivative works that are provided with these compilers Fairly but for whatever you wish as long as you agree that Bill Buckels and the other Copyright holders and contributors have no warranty or liability obligations whatsoever from said use. None of this may be bundled with a product for resale without additional permission from the applicable Copyright holders.

Contents

1. TUTORIAL INTRODUCTION T0 AZTEC C65....5
   1.1 Getting Started....5
   1.2 Configuring the SHELL....6
   1.3 Two Drive Environment....7
   1.4 Creating the Program....8
   1.5 More SHELL Goodies....10
   1.6 C65 and CCI, The Speed Versus Size Dilemma....10
   1.7 Compiling and Assembling....11
   1.8 A Few Utilities....12
   1.9 Linking with the Library....13
   1.10 Running the Program....15
   1.11 More Choices....16
   1.12 Going to the Source....17
   1.13 Where to Go From Here....18
2. THE SHELL....20
   2.5 General Use....20
   2.6 Built-in Commands....21
      2.6.1 boot....22
   2.6.2 bye....22
      2.6.3 call....22
      2.6.4 cat....23
      2.6.5 cd....24
      2.6.6 ce....24
      2.6.7 cp....24
      2.6.8 load....25
      2.6.9 lock....25
      2.6.10 ls....25
      2.6.11 maxfiles....26
      2.6.12 mv....26
      2.6.13 rm....27
      2.6.14 run....27
      2.6.15 save....28
      2.6.16 unlock....28
   2.7 Batch Facilities....28
      2.7.1 loop....29
      2.7.2 set....30
   2.8 Configuration....30
      2.8.1 Keyboard....31
      2.8.2 Screen....31
      2.8.3 Printer....32
3. PROGRAMS....34
   3.1 C65 Native Code Compiler....34

3.2 CCI Pseudo-code Compiler....40
3.3 AS65 6502 Assembler....41
3.3.1 Overview....41
3.3.2 Syntax....42
3.4 ASI Pseudo-code Assembler....44
3.5 LN Linker....45
3.6 MKLIB....47
3.7 VED Screen Editor....49
3.8 ARCH Source Archive Utility....51
3.9 OD Hex Dump Utility....52
3.10 CMP Byte for Byte File Compare....53
3.11 NM Name List Generator....54
3.12 TABSET....55
3.13 CONFIG....55
3.14 LDEV....55
4. LIBRARIES....57
4.1 Introduction....57
4.2 Summary of Library Functions....59
4.2.1 Standard I/O....59
4.2.2 System I/O....60
4.2.3 Utility Routines....60
4.2.4 Math Routines....61
APPENDIX A: Compiler Error Codes....63
APPENDIX B: The Graphics Disk....66
B.1 FILES ON THE GRAPHICS DISK (DISK7)....66
B.2 GRAPHIC FUNCTIONS....66
B.2.2 plotchar....68
B.2.3 circle....68
B.2.4 line routines....69
B.2.5 page....70
B.2.6 color....70
B.2.7 plot....71
B.2.8 mode....71
Appendix C: DISKS....73
Appendix D: Mini Manual Credits and Attributions....74
D.1 Rubywand 74
D.1.1 The Compiler....74
D.1.2 Other Attributions....76
D.2 Manx Software Systems....76
e.o.f. Rubywand 13Jan2001 amdg....77

INTRODUCTION

1. TUTORIAL INTRODUCTION T0 AZTEC C65

1.1 Getting Started

Congratulations on choosing the Aztec C65 compiler from Manx Software Systems. This part of the manual contains sections on installing and configuring the SHELL command processor to your Apple. It then proceeds step by step through the creation, compiling, linking and running of a test program. On the way, it will introduce important parts of the SHELL which give a very UNIX-like environment on a small machine. The remainder of the manual is more of a reference guide and provides more detailed information on the individual pieces and procedures.

The Aztec C65 system is shipped on either three reversible or six single sided diskettes. These diskettes should be copied to six other single sided diskettes before being used These diskettes have been initialized with a special program which allows files to be stored on tracks 1 and 2 which are normally reserved for DOS. Therefore, the best way to copy them is to use the COPY or COPY A program which is supplied on the DOS 3.3 master. The originals should then be stored in a safe place in case they are needed again.

To boot the SHELL, first boot DOS 3.3 from the DOS 3.3 system master supplied with your Apple. The disks supplied by Manx cannot be booted. Then, insert the disk labelled STARTUP (Disk1) into drive 1 and type:

BRUN SHELL

Note: This set includes a bootable Disk 1 which automatically BRUNs SHELL.

SHELL is a binary program which contains the new command processor, the pseudo-code interpreter, and part of the library. It will automatically move itself to the appropriate places in memory. For more information, consult the memory map in the SHELL section of this manual.

The SHELL will display the message:

```
APPLE ][ SHELL 2.4
COPYRIGHT (C) 1983
BY MANX SOFTWARE SYSTEMS
```

on the screen and the prompt:

-?

Following the prompt should be a solid cursor. At this point, the SHELL is up and running, and assumes that the Apple it is running on is a normal 'bare bones' Apple II without lower case keyboard inputs.

Note: This changed with version 2.4. It is pre-configured to expect and recognize upper and lower case keyboard inputs. To use on an Apple II without lower case, use the CONFIG option as discussed below.

In 'bare bones' config, the ESC key acts as a caps lock/unlock key, lower case is displayed as normal text, and upper case is displayed as inverse video. Try typing some characters. They should appear as normal video characters. Now, press the ESC key. The first thing you should notice, is that the cursor is now flashing. This signifies that you are in CAPS LOCK mode. Try typing some characters now.

They should appear as inverse video characters, which indicates that they are upper case. If you are using a keyboard with full upper and lower case capability, such as the Apple IIe, you will need to type the characters as upper case. This is necessary since the SHELL is delivered configured for a basic Apple II. We will discuss how to take advantage of additional capability shortly. On the IIe, simply make sure that the CAPS LOCK key is engaged.

Type ^X (CONTROL-X) to cancel the line that you typed. (Note that control characters will sometimes be displayed in this manual as a caret followed by the appropriate character.) There are a number of other control characters which have special meaning. The full list is discussed in the SHELL reference section on console I/O.

The important ones are:

^H (also the left arrow key) which is used to backspace over the last character typed.
^X to cancel the current line of input.
^S to stop and restart output to the screen.
^C to abort a program and return control to the SHELL.

Under the SHELL, the command to catalog the disk is "ls". Try typing it now followed by a return to see the files on the ST ARTUP diskette. The SHELL normally assumes that commands are typed in lower case. If you typed "LS", the SHELL tried to find a file with the name "LS" to run, and gave an error message when it didn't find it. Hit the ESC key to get out of CAPS LOCK and try it again. The "ls" command is "built- in" to the SHELL A full list of the built-in commands can be found in the SHELL reference section.

1.2 Configuring the SHELL

Up to this point, the SHELL has ignored any peripherals or options which you might have added to your machine. To make use of these features, the SHELL must be configured to the exact system which you are using. This is done by using the CONFIG program which is also on the STARTUP disk.

To run the CONFIG program, simply type:

config

followed by a return. Note that unlike DOS, you don't need to type RUN or BRUN to execute programs. Simply the name of a file will cause it to be loaded and executed.

When the CONFIG program has been loaded, it will display a startup message and ask a series of questions about the machine you are using and the peripherals installed. Most questions can be answered with a simple 'y' or 'n'. A more detailed discussion of the CONFIG program and the meaning of some of the questions can be found in the CONFIG reference section.

At one point in the program, it will ask if you are using an 80-column video card. If you answer yes, it will ask about specific cards that it has tables for. If the card you are using is not in this list, you must provide information from the card's manual. For the purpose of this introduction, you may wish to cancel the CONFIG program and perform the configuration later after reading the CONFIG reference section. In the meantime, the default configuration should suffice till then.

When the configuration is finished, the program will ask if you wish to store that configuration. If you answer 'n', only the current memory version of the SHELL will be altered.

1.3 Two Drive Environment

One of the nice features of the SHELL is it's use of two drive disk systems. To illustrate this, insert a DOS initialized diskette into drive two. To catalog drive 2, type:

ls d2

This is different from the DOS way of doing things in two ways. The command name "ls" must be separated from its argument by at least one space, not a comma. The second thing that is different is that this command does not make drive two the active drive. Typing the "ls" command by itself will still give the catalog for drive one. To change the active drive, the "cd" command must be used. Type:

cd d2

to change the active drive from drive one to drive two. Now drive two will be the active drive until another "cd" command is given, or the system is rebooted.

The other nice feature for multi-drive systems is the concept of an execution drive. For example, try typing:

config

Note that the drive light on the active drive will go on as the SHELL tries to find the program. Assuming that you don't have a program named CONFIG on the scratch disk,

the SHELL will then automatically check the current execution drive. In this case the current execution drive will be drive one, and the CONFIG program will be loaded The current execution drive can be changed by using the "ce" command in a manner similar to the "cd" command.

For the rest of this introduction, it will be assumed that the current execution drive is drive one, and that the current data drive is drive two. After cancelling out of the CONFIG program by typing ^C, type the following just to be sure:

ce d1
cd d2

Then replace the STARTUP disk in drive one with Disk3 labeled C65. (The CCI disk is Disk4.)

1.4 Creating the Program

The C65 disk contains the 6502 C compiler, the 6502 assembler, and several utility programs. In the following paragraphs, we will use the VED screen editor to create a test program which we will then compile, assemble and link with a library. The result will be a file which we can then execute. We will also make use of some of the other utilities as 'well. The program which we will write gives a useful demonstration of how arguments passed to a program are accessed by the program. The following is a listing of the program:

```
main(argc, argv)
int argc;
char *argv[];
{
  register int i = 1;

  printf("Program <%s> has %d arguments\n", argv[0], argc-1);
  while (--argc) {
    printf("Arg %d = <%s>\n", i, argv[i]);
    i++;
  }
}
```

As can be seen, the program prints its name, which is the first argument, and the number of arguments. Since the number of arguments includes the program name, argc-l is used as the number of real arguments. Then, each argument is listed on a separate line. The first step is to create the source program using the VED screen editor. Type:

ved args.c

VED will be loaded from the current execution drive, and will try to find "args.c" on the disk. When it doesn't find it, it will say so and will start with an empty document. Note that the screen should look like:

"args.c" line 1 of 1
-
-
-

The cursor should be on the second line, and a single '-' on all the remaining lines. The '-' indicates that the line is after the end of the file. If the screen does not look this way, there is something wrong with the way that your SHELL is configured Refer to the CONFIG reference section before proceeding further.

VED has two modes, command and insert. Normally, VED is in command mode. For a list of most of the commands available, try typing a question mark without a return. The screen should clear, and the list should appear. Pressing the return key should repaint the screen with the document being edited To enter insert mode, simply press the 'i' key. On the status line, the <INSERT> mode indicator should appear. This will always be there when in insert mode.

At this point, type in the test program, using the left arrow key to correct any mistakes. The indentation in the program is produced by using a tab character. The tab character width is defined by the SHELL and defaults to four. It can be changed using the TABSET program discussed in the PROGRAMS section of this reference manual.

If you are using a standard Apple II keyboard, you will need help to produce some of the characters. To get the '{', type ^A. To get the '[', first press the ESC key to go to CAPS LOCK mode and then type ^A. If you have installed the SWSKM (single wire shift key mod), and configured the SHELL for it, then use shift ^A to get the '['.

The following table lists the other mappings you will need. The capitalized control characters must be typed with the CAPS LOCK on or the shift key down if the SWSKM is installed.

```
^a   (
^A   [
^I   tab (the right arrow key on Apple II's may be used as well)
^r   }
^R   ]
```

VED expects an ESC character to end the insert mode. If the ESC key is being used as a CAPS LOCK key, the AQ key will produce an ESC character instead Once out of insert mode, the cursor can be moved around using the space bar to move right and the left arrow to move left. To move a number of characters to the right or left, type the number of characters to skip followed by the space or backspace. To move to the beginning of the next line, use the return key. Similarly, use the ‘-‘ key to move to the beginning of the previous line. Characters can be deleted by placing the cursor on the character and pressing the 'x' key. Characters can be inserted by placing the cursor at the insertion point and pressing 'i' to enter insert mode.

file modified - use q! to override

This message will appear whenever you try to exit VED after making a change without writing the file out. To exit without saving the changes made, type ":q!" followed by return.

1.5 More SHELL Goodies

At this point, the SHELL prompt should be back. To examine the file you created, you may either use VED again, or type:

cat args.c

to display the file on the screen. This uses the built-in SHELL command, "cat", which opens it's arguments one by one and copies them to the standard output.

If you have a printer card installed and configured correctly, you can print the file with the following command:

cat args.c > pr:

This introduces another feature of the SHELL, I/O redirection. Under the SHELL, when a program is invoked, it has three pre-opened channels of communication. These are usually referred to as the standard input, output and error. Normally, the standard input channel is connected to the keyboard, while the standard output and error channels are both connected to the screen. However, by using the special characters '<' and '>', the standard input and output can be "redirected" to other devices.

Thus in the above examples, the "cat" command opens the file specified by the argument and reads the contents of that file and writes them to the standard output. In the first case, that was the screen. By using the "> pr:" in the second example, the SHELL switched the standard output to "pr:" which is the name of the printer device. The name of both the keyboard and screen is "kb:". We will say more about I/O redirection later.

1.6 C65 and CCI, The Speed Versus Size Dilemma

Now that we have our C source program, the time has come to compile it. The Aztec C65 system actually comes with two C compilers. The first compiler, C65, produces 6502 machine code, while the second compiler, CCI, produces a pseudo-code that must be interpreted. Because of the architecture of the 6502 microprocessor, there are advantages to both.

The 6502 microprocessor is completely restricted to dealing with single bytes at a time. Addresses and numbers larger than 256, on the other hand, are two bytes in size. As a result, the 6502 machine code generated by C65 tends to be larger than programs

produced for machines which have better facilities for handling 16-bit quantities. As an alternative, the pseudo-code C compiler, CCI, produces machine language for a theoretical machine with 8, 16 and 32 bit capabilities. This machine language is interpreted by an assembly language program that is about 3000 bytes in size.

The effects of using CCI, are twofold. First, since one instruction can manipulate a 16 or 32 bit quantity, the size of the compiled program is generally more than fifty percent smaller than the same program compiled with C65. However, interpreting the pseudo-code incurs an overhead which causes the execution speed to be anywhere from five to twenty times slower.

Note: The note above refers to CCI compiled (Shell) programs that do not mix native code into speed-critical portions. But from that time to this execution speed has improved immensely. This manual was written when accelerators weren't around much. Really fast Apple II accelerators like the ZIP chip came even later. Today, emulators like AppleWin with fast speed settings, and fast "virtual" disk image access, can run Shell programs very quickly; With the benefit of very tiny compiled Shell programs to save the limited disk space on a DOS 3.3 disk, and the benefit of the Shell's real command line and other features like redirection and compatibility with shell scripting not available in DOS 3.3.

The dilemma appears obvious: speed versus size. For most applications, hopefully, the resolution is obvious. If the program is small, there should be no problem using C65. If the program is large and the speed of execution not critical, use CCI. If the program is large and execution speed important, there are at least three solutions.

First, code extremely time critical parts of the program directly in 6502 assembly language. This is typically necessary in applications such as real-time graphics, where the overall program is written in a higher level language, but the extremely time-critical portions are written in assembly. A second approach, similar to the first, is to compile just the time critical routines with C65 and the remaining routines with CCI.

Both compilers and assemblers have been designed so that the object modules produced by each may be combined together into one binary program.

A third possibility is to use the overlay facility provided with this system. Overlays allow portions of a program to be loaded from a disk when they are needed, and then to be "overlaid" with other portions. Using this technique, the size of a program need only be limited by the size of the disk you are using. Finally, any combination of the above methods may be used to achieve a satisfactory balance of size and execution speed

1.7 Compiling and Assembling

The examples and discussion which follow are restricted to C65, but basically apply to CCI as well. The simplest way to use C65 is to type:

c65 args.c

The compiler will be loaded from the execution drive and will display the version number and the copyright message. It then translates the source file into 6502 assembly language. The assembly language is placed in a file called "$TMP.$$$" which will be deleted later by the assembler. After the compiler finishes, the 6502 assembler, AS65, is automatically loaded. AS65 assembles the assembly language in "$TMP.$$$" and places its output in a file called "args.rel". The type of the ".rel" file is 'R' indicating that it is a relocatable object file. When the assembler finishes, it deletes the temporary file, "$TMP.$$$". At this point the compile is finished.

While the source is being compiled, if any errors are detected the line containing the error will be displayed, along with the line number and the error number. Refer to the error summary in the appendix to translate the error number. If there is an error, use VED to edit the file. To move the cursor to the line with the error, type the line number followed by a 'g'. Correct the error, write the file, and recompile it.

If you wish to compile without assembling, then typing:

c65 -a args.c

will compile the program and produce an assembly language text file called "args.asm". This file may be edited or printed as desired When this option is used, the assembler is not automatically executed. To produce a relocatable object file from "args.asm", type:

as65 args.asm

AS65 will place its output in a file called "args.rel". Note that in this case, AS65 will not delete "args.asm" when it is finished.

1.8 A Few Utilities

The relocatable object file produced by both assemblers is in a special binary format. If you "cat" the file to the screen, the result will be visual garbage. Instead. to look at the contents of a non-text file, there is a utility program called OD. This is not a built-in command and must be loaded from the disk. To execute the program, type:

od args.rel

The program will open the file args.rel and display in hex the value of each byte in the file. If the byte is also a displayable character, it will be displayed at the right of the hex values. Non-displayable characters will be displayed as a period. The display can be temporarily stopped and restarted by using the ctrl-S key. The program can be aborted by typing ctrl-C. OD can be used to dump the contents of any file, text, binary, basic, or relocatable object Try it on "args.c".

A second utility, NM, works only with relocatable object modules. This utility performs two functions. First, it can display the size of the code and the data which is contained in an object file. This is useful since the physical size of an object file does not directly reflect the size of the code and data which will be produced when it is converted to absolute binary form. To see the size of the code produced by the "args.c" program, type:

nm -s args.rel

The result should be about 210 bytes if you used C65, and 96 bytes if you used CCI.

The second function of NM is to display the names and offsets, if known, of all labels defined in a module. This is mostly useful when building libraries. It is possible to determine what labels are defined within this module and which are yet to be defined. The various options for the output can be found in the PROGRAMS reference section. Typing:

nm args.rel

will show that the "main_" function is defined in this file, and that several functions are undefined, including "printf_".

1.9 Linking with the Library

Note: Aztec C65 .REL files (discussed below) are in their own format and are saved as binary files (with a .REL extension) on a DOS 3.3 disk and not as a DOS 3.3 REL file type 'R'. The REL file format used with the CII compiler (this vintage) was consistent across Aztec C's respective native-mode and cross-compilers for the Apple II, Commodore 64, and CP/M, which also has its own Digital Research (DR) REL file format which differs from the various REL formats found on the Apple II. By the time ProDOS came around Apple Computer applied many restrictions to file types, but never implied restrictions to the ProDOS REL file type simply describing a REL file as "Relocatable code" although some folks (like APDA in their standard and Apple themselves in manuals like the EDASM Manual) assigned the type exclusively. By the time the Aztec 65 version 3.2b Apple II native compiler was released (the last Apple II Aztec C native Compiler) in 1987, the .REL extension had been replaced with a .O extension like their MS-DOS C86 version. But Aztec C was inconsistent with its naming and the 3.2b Apple II cross-compiler for MS-DOS used the .R extension for their object files, and calling the same program by different names for different platforms (LN in this compiler is called LN65 in the MS-DOS Apple II cross-compilers) making manuals like this difficult to decipher at times… and to make things even more confusing the object file formats between the 3.2b compiler and this compiler were different too. So to recap, for the purposes of reading and understanding this manual, the REL format is an extension and not a DOS 3.3 file type. Object files for the SHELL have a .INT extension which could be confusing too! You can also use your own naming by –o (over-riding output default naming). Sheesh!

Both assemblers translate assembly language into a format called relocatable object format. This format is designed to allow the program module to be converted into absolute data which will be loaded and run at a specific address in memory. This becomes particularly important when the final program consists of several modules compiled and assembled separately. As will be seen, this is true of almost all programs.

For example, assume that a program consists of two modules, "main.rel" and "subs.rel". Assume, also, that "subs" contains several functions to be called from "main". Since the two modules are compiled and assembled separately, there is no way for "main" to know where "subs" is going to be in memory. Even if "main" did know the address of the beginning of the "subs" module, it has no way of knowing the size of each function in that module.

It is possible that one could give all the information needed when compiling and assembling "main" to directly produce a binary image. This is only practical if the amount of information needed is quite small However, most C programs make use of a number of functions supplied with the compiler. These functions are usually kept in individual modules so that functions not used by the program are not included.

The number of these functions make it totally impractical to produce any kind of direct binary output. The solution is the relocatable object format and a program to link object modules together, the Aztec linker, LN.

LN combines any number of object modules together and produces a binary file in the standard Apple DOS "BRUN" format. LN will also indicate if anything is missing. For this example replace the C65 disk with the LIB65 disk (Disk5) and type:

ln args.rel

In this case, LN will attempt to produce a binary file from "args.rel". However, since the "args" program makes reference to several functions which are not defined in the "args" module, the linker will give error messages to that effect.

Supplied with the Aztec C65 system, is a large set of subroutines which perform many different functions. A large percentage of these routines are used to perform input and output operations, since the C language has no inherent mechanisms for doing I/O. A complete list of these functions and a description of each can be found in the LIBRARIES section of this reference manual.

To simplify the process of selecting the correct routines to be linked with a particular program, it is possible to combine a number of routines into a single file, called a library. The format of a library is designed so that individual modules can be read from it without reading all the modules. In addition, the linker, LN, will search a library and only use those modules which satisfy references made in other modules that it has processed.

Thus, to correctly link the "args" program, type:

ln args.rel sh65.lib

In this case, the linker will read the "args.rel" file and make a list of all undefined symbols. Then, it will check the library (note that LN looks for modules or libraries on both the data and execution drives automatically) for any modules which contain the proper symbol If it finds one, it will read that module from the library. If there are any undefined symbols in that module, they are added to the list.

This process continues until the end of the library is reached. If there are still unresolved symbols in the list, they are displayed in error messages and the link aborted. If all the unresolved symbols get matched up with corresponding routines in the library, then the linker proceeds with combining all the object modules together into one binary program.

If the link was successful, there will be a binary file called "args" located on the current data disk. LN will call the output file the same name as the first object module argument To specify a different name, LN can be used with a "-o" option as follows:

ln -o testprog args.rel sh65.lib

which will place the output in a file called "testprog" instead.

And that's all there is to it! In summary, to turn "args.c" into the program "args" takes only two commands:

c65 args.c
ln args.rel sh65.lib

1.10 Running the Program

Now that the program has been compiled, assembled and linked, it's ready to be run. All that has to be done is to type:

args these are some args

which will display:

Program <args> has 4 arguments
Arg 1 = <these>
Arg 2 = <are>
Arg 3 = <some>
Arg 4 = <args>

Note that the SHELL automatically parses the command line and breaks it up into pieces which are separated by blanks. To type an argument which contains a blank, it is necessary to enclose the whole argument in double quotation marks. For example, try:

args "arg one" "arg two"

To save the output of the "args" program in a file, we can use the I/O redirection capability of the SHELL. The printf() routine that we used in "args" sends its data to the standard output which can be redirected, as in:

args one two three > args.out
cat args.out

The first line calls "args" with three arguments. The '>' and all following information is directed to the SHELL and is not passed to the program. The file "args.out" now contains the output that would have gone to the screen. I/O redirection can be used to redirect I/O to or from disk files, or the devices "kb:" and "pr:".

1.11 More Choices

There are basically two libraries supplied with the Aztec C system. One contains the transcendental math functions and the floating point emulation routines. The second contains all the other routines. When linking, the FLOAT library need only be specified if floating point is used somewhere within one of the modules. If floating point has been used, and the program is linked without the FLOAT library, there will probably be a number of unresolved references. In particular, the symbol, ".fltused", indicates that floating point was used at some point. This is an example of a case where the NM program could be used to determine which modules declared ".fltused" as undefined.

When linking with the FLOAT library, it should be placed before the regular library in the argument list. For example:

ln -o flargs args.rel flt65.1ib sh65.lib

will create a binary program called "flargs" which contains the floating point emulation routines.

Although there are only two basic libraries, there are a number of different flavors of each. The FLOAT library comes in only two flavors, FLT65.LIB and FLTINT.LIB. Both libraries contain the same functions, but all the C language routines in FLT65.LIB have been compiled with C65, while those in FLTINT.LIB with CCI.

The regular library also comes in a C65 version and a CCI version. However, there is another distinction as well. One version of the regular library is only useful when creating programs that will run while the SHELL is in memory. The other version is designed to allow programs to run directly under Apple DOS with or without the SHELL. The second version is called the STAND-ALONE library.

The SHELL libraries are called SH65.LIB and SHINT.LIB which correspond to the C65 and CCI versions respectively. Likewise, the STAND-ALONE libraries are called SA65.LIB and SAINT.LIB. All the libraries compiled with C65 are on the disk labeled LIB65, while the disk labeled LIBINT contains the others.

Note: This is not quite true. In order to fit the files onto the diskettes, the STAND-ALONE libraries are swapped around. So, you have...

Disk5 "LIB65"
SH65.LIB for C65 Shell lib
FLT65.LIB for C65 Float lib
SAINT.LIB for CCI Stand-alone lib

Disk6 "LIBINT"
SHINT.LIB for CCI Shell lib
FLINT.LIB for CCI Float lib
SA65.LIB for C65 Stand-alone lib

Note: This business of STAND-ALONE does not mean that everything called STAND-ALONE (SA prefix) can run in "RAW" DOS 3.3. Only two libraries can do that; SA65.LIB and FLT65.LIB. Only two libraries are mostly PCODE and lever the Shell's built-in calls; SHINT.LIB, which produces the smallest executables which run only in the Shell INTerpreter, and the other library, SAINT, is a hybrid but can only be used for programs that run in the Shell. To make things even more confusing, native mode modules or libraries can be linked with Shell libraries (with varying degrees of success). My recommendation after years of using this vintage of Apple II compiler is the general rule of linking your shell programs to SHINT.LIB and your "RAW" programs to SA65.LIB. Avoid running your RAW programs in the Shell unless you don't do much text screen manipulation. The Shell does well with its own but fails on many "RAW" calls. Example programs that do both are in the Aztec33 bundle which repackages this compiler with a cross-compiler for MS-DOS of the same vintage. And now you are likely horribly confused! But you have options!

G.LIB (the graphics library) is a "RAW" library which works both in the Sheell and in RAW DOS 3.3 but special rules apply when writing for the Apple II's Graphics Screen to avoid clobbering it with your program. Atztec 33 has samples for that too.

The differences between the STAND-ALONE library and the SHELL library are discussed in the LIBRARIES section of this manual. The FLOAT libraries may be used stand-alone or with the SHELL.

1.12 Going to the Source

The source to most of the library routines, some of the utility programs, and parts of the SHELL, are included with the Aztec C system. These text files are collected together in a set of binary files called archives. Placing the files in archives allows more efficient use

of the disk space. Replace the disk in the current execution drive (drive 1) with the disk labeled ARCHIVES and type:

cp progsrc.arc,d1 progsrc.arc

The "built-in" command, "cp", will copy the file "progsrc.arc" from drive one to the current data drive (drive 2). Now type:

arch -l -o progsrc.arc

The "-l" (lower case "L") option tells the ARCH program to list the names of the files in the archive. The name of the archive is specified by using the "-o" option. ARCH will list the name and size of each file in the archive. To extract one of the files, type:

arch -x -o progsrc.arc tabset.c

The "-x" option tells ARCH to extract the file names which are passed as arguments. Thus, more than one name may be specified at a time. That is also why the "-o" option is necessary to tell ARCH which argument is the archive itself. If the "-x" argument is specified with no filenames, then all the files in the archive are extracted.

Included with this manual should be a release document which describes the contents of each archive.

1.13 Where to Go From Here

Well that about covers the basics. The rest of this manual is devoted to giving more precise technical information on a number of different topics. The major sections and their contents can be summarized as follows:

SHELL	- commands and features
PROGRAMS	- options and use of each program
LIBRARIES	- calling sequence and function
TECH INFO	- a variety of information

Familiarity with the sections on the SHELL and options to the programs is highly recommended. In the beginning of the libraries section, there are several sheets which provide a summary of the library functions and their arguments. A copy of these sheets along with a copy of the compiler error codes can be found as the last pages of this manual and can be used as a handy reference. The last section contains a number of different documents which provide information on a variety of topics, including overlays, floating point format; ROMable code, device drivers, stand-alone use and others.

THE SHELL

2. THE SHELL

2.5 General Use

The simplest form of a SHELL command is the name of a function followed by a carriage return. A SHELL command may either be one of the built-in utilities or the name of a binary or text file which resides on disk. The following is a list of the built-in functions available with the SHELL. If a file has the same name as one of these functions, the SHELL will not execute that file, but will execute the built-in function instead.

boot	**cp**	**mv**
bye	**load**	**rm**
call	**lock**	**run**
cat	**ls**	**save**
cd/ce	**max files**	**unlock**

These commands are all specified using lower case. A complete description of each command can be found in the Commands section.

Binary programs which are normally run using the DOS 'BRUN' command can be loaded and executed by simply typing the name of the file followed by a carriage return. The first two words of binary files which contain executable programs contain the load address and length in bytes of the memory image. These are used to load the program into memory. The SHELL 'load' command can be used to load the image into a different section of memory much the same as the DOS 'BLOAD' command.

Text files containing a series of SHELL command lines can be executed by simply typing the name of the text file followed by a carriage return. All SHELL input is then taken from that file until the end is reached For more information see the Batch Facilities section.

Some built-in SHELL utilities as well as binary programs produced using the Aztec C compiler system require or allow arguments to be specified when the command is executed These arguments are placed on the same line as the command name separated by spaces. An example of this is the SHELL 'lock' command Under Apple DOS, if it is desired to lock several files, the DOS 'LOCK' command must be given once for each file. To lock several files using the SHELL, the user would type something like:

lock test1 test2 test3,d2

to lock files "test1" and "test2" on the current drive and "test3" on drive two.

Because arguments are separated by spaces, file names containing spaces must be enclosed in double quotes to enable the SHELL to distinguish the single name from two names. For example, to unlock a file called "testprog", the user would type:

unlock "testprog"

Double quotes should also be used around file names used as commands if the name contains any blanks.

The final feature of the SHELL which will be discussed is the ability to redirect the standard input and/or output of a program to a file or a device. Normally the standard input and output of a program are connected to the keyboard and screen respectively. The user may redirect either or both of these connections to a file or a device such as a printer. This is accomplished by using the special character '<' for input and '>' for output.

As an example, to place the output of the NM command, which produces a symbol table from an object file, into a file for later perusal, type:

nm objfile > listing

The namelist will not be printed to the screen, but to the file 'listing' instead. The SHELL also pre-opens a second channel to the screen
called the standard error output. This channel cannot be redirected.

2.6 Built-in Commands

This section describes the commands which are built into the SHELL program itself. Each SHELL command will be listed along with a description of
its use and its function. All commands are specified as being lower case. File names may be typed with either upper or lower case letters, however they will all be mapped to upper case for compatibility with Apple DOS. File names may contain blanks, but to distinguish arguments from the parts of the file name, the entire name must be enclosed within double quotes.

In the following discussions, the concept of current data drive and current execution drive are used Under DOS, the last drive accessed is considered the current drive. Under the SHELL, the current data slot, drive and volume must be explicitly changed by the user using the "cd" command at any point where an optional slot, drive or volume parameter may be given. If any are not specified, they will default to the current data value respectively. The examples given for specific commands should clarify this point.

In general, all arguments to SHELL commands and to utility programs are separated by blanks. Arguments in square brackets are optional and most commands allow more than one file name per command line. In the following descriptions, any reference to a file name is assumed to include the optional slot, drive and volume parameters.

2.6.1 boot

boot n

Does a jump to (slot number n) address $Cn00. (Usually to reboot a drive controller installed at slot n (4, 5, 6 or 7).)

Example:

boot 6

Causes the floppy disk to reboot

Note: The boot command is really the equivalent of typing "PR#" in a BASIC program; boot 6 will reboot a floppy drive system with the floppy controller in slot 6. If you have a Microsoft CP/M Softcard in Slot 4 or 5, or an Applicard in slots 4, 5, or 7, this command can be used to get to CP/M from DOS 3.3 by typing boot followed by the respective slot number. Since hard disk systems aren't really supported by the DOS 3.3 filing system, boot 7 for a hard drive would be unlikely.

Doing a boot 3 with an 80 column card installed which has the same effect as a jump to $C300 messes-up the shell's text screens and creates double spacing. Configuring the shell to 80 column mode and letting the shell take care of text screens is the only alternative, and a jsr or a jump to $C300 should never be done in a shell program if you want your text screens to work afterwards. The shell does its own thing when it comes to screen control and you are best to avoid any of the "RAW" jumps and jsrs to manipulate text screens in shell programs. The shell has routines for that, and the text screen cursor control and clearing the screen are supported by the shell's internal terminal routines. To recap, this command is useful to reboot your floppy (boot 6) including in a shell script that is not being redirected.

2.6.2 bye

bye

Does a jump to the Apple machine language monitor at location $FF65. Re-entry to the SHELL is through $3D0 or by hitting RESET on systems with the autostart ROM.

2.6.3 call

call addr

Performs a "jsr" to the address given. If addr is preceded by a '$', it is interpreted as hex, otherwise as decimal.

Examples:

call $800
call -151

The first example does a "jsr" to hex 800, while the second calls the monitor.

Note: Careful using this one. It probably works ok to run a little bit of code loaded into where the shell expects, like at $800. It mucks-up on calls to routines like catalog at $a56e. Equivalent shell commands can be used in some cases; i.e. ls maps to catalog and works properly. In your own shell programs, using the runtime library calls supplied with Aztec C like the catalog() call is a better alternative, but in a "RAW" DOS 3.3 program you are free to do what you want. However, those "RAW" programs that do what they want don't always run properly in the Aztec C65 DOS 3.3 shell. The newer ProDOS Shell is more forgiving.

2.6.4 cat

cat [file1] [file2] ...

Concatenates the named files to the standard output. If no files are specified, input is taken from the standard input. This is the quickest and easiest way of looking at a text file.

Examples:

cat test1 test2,d1
cat test1 test2,d1 > test3
cat > pr:
cat kb: > pr:
cat > myfile.txt

The first example displays "test1" from the current data drive on the screen immediately followed by the file "test2" located on drive one. The second example creates a new file called "test3" containing the two files "test1" and "test2". The third example reads a character from the standard input and writes it to the device "pr:" which is the printer. The fourth example is equivalent to the third.

Note: One of the most useful variations of the cat command was never included in the orginal manual. I have placed this as example 5 in the examples above. By redirecting to a text file and then pressing ctrl-c (ctrl-break) when done, you can create shell scripts and other 7 bit sequential ascii files without the need of an editor. The gotcha' here is that you can't miss a typo on a previous line or insert a line above the current line, because all you are doing is copying stdin to stdout. You can have blank lines in these files and pretty much any character that the Shell console driver accepts.

2.6.5 cd

cd sn,dn,vn

Change the current data slot, drive and/or volume. Any or all of the three parameters may be changed. Those not specified will remain the same. If a volume number is specified., it will be checked whenever a file is opened A volume number of zero, however, will match any disk.

Examples:

cd s6,d1,v0
cd d2

The first example changes the current data disk to be slot six, drive one, and any volume. The second example changes from whatever the current drive was to drive two. The slot and volume remain the same.

2.6.6 ce

ce sn,dn,vn

Change the current execution slot, drive and/or volume. Execution parameters are used when loading and running a particular binary program or SHELL file. If the name includes a specific reference to a slot, drive or volume, that parameter is used If there is no reference as to which device holds the file, the current data disk is searched and if the file is not found there, then the current execution disk is checked This allows all utility programs to reside on a different disk than the one being actively used.

ce s6,d2,v0
ce d1

The first example changes the current execution disk to be slot six, drive two, and any volume. The second example changes from whatever the current drive was to drive one. The slot and volume remain the same.

2.6.7 cp

cp file1 file2

Copies files from the specified device to file2. Note that file2 will be overwritten if it already exists.

Examples:

cp test oldtest

cp test,d1 test

The first example makes a copy of "test" on the same disk called "oldtest". The second example assumes that drive one is not the current data drive and copies the file "test" from drive one to the current drive.

2.6.8 load

load file [aN] [lN]

Loads a binary file into memory. If the starting address and/or length are not specified, they are taken from the first two words of the file. After loading, the start address and length are displayed on the screen. These values are remembered for use in the save and run commands. If N begins with a '$', the value is interpreted as a hex value otherwise as decimal.

Examples:

load tabset
A=0800 L=12F2 (read from the binary file header)

load tabset a$2000
A=2000 L=12F2 (length read from the binary file header)

The first example loads the tabset program into memory. The shell displays the load address and length. The second example loads the tabset program into memory at address hex 2000.

2.6.9 lock

lock file 1 [file2] ...

Lock the file on the specified slot, drive and volume. If any of slot, drive or volume are not given, they default to the current data values.

Examples:

lock test1
lock test1 test2 test3,d2

The first example locks file test1 on the current data disk. Example two locks files test1 and test2 on the current data disk and locks file test3 on drive two of the current data slot and volume.

2.6.10 ls

ls [sn,dn,vn] ...

Perform the catalog function on the specified slot, drive and volume. This command defaults to the current data slot, drive and volume. If more than one is specified, they will be cataloged in order. The SHELL will wait for a key to be pressed between different catalogs. Unfortunately, the output of ls cannot be redirected.

Note: 3 example programs provided with the Aztec33 distribution provide different methods of creating text files of directory lists. Two of these are for the shell; one (called DLIST) uses the C65 runtime catalog() function and the other (called LS33) uses a lower level C65 function called rwts() (Read Write Track Sector), which is also in the Aztec C65 runtime library. Output from LS33 can be redirected and options are provided for search criteria based on file type or extension or both. LS33 also creates shell command scripts as an output option. The other two programs (DIR33 and DLIST) write text files. LS33 and DIR33 (its "RAW DOS 3.3" equivalent) both provide output in 7 bit plain text or DOS 3.3 text. But the Shell's ls command is simply a wrapper for their runtime catalog() function, and although necessary it cannot be redirected to create lists and it cannot be scoped to provide a selective listing:

Examples:

ls
ls d1 d2

The first example does a catalog of the current data slot, drive and volume. The second example catalogs drive one and then drive two of the current data slot and volume.

2.6.11 maxfiles

maxfiles n

Allocates n buffers for open files. This command is similar to the DOS 'MAXFILES' command. It specifies the maximum number of disk files which may be open at anyone time. When the SHELL is initialized, the value is defaulted to 3.

Example:

maxfiles 4

For an application which will have four disk files open, maxfiles is set to four.

2.6.12 mv

mv [-f] file1 file2

Moves file1 to file2. If the slot, drive, and volume of file1 are the same as that of file2, file1 is simply renamed as file2. If they are different, file1 is copied to file2 on the specified device and file1 is deleted. If file2 exists, an error message will be printed. If the '-f' option is given, no error message will be given and file2 will be removed first.

Examples:

mv test foo
mv -f test foo
mv test test,d2

The first example simply renames the file "test" as "foo". The second example deletes the file "foo" and then renames "test". The last example copies the file "test" from the current data drive to drive two and then deletes "test" from the current drive.

2.6.13 rm

rm file1 [file2] ...

Delete the specified file or files. If a file is locked, a message is displayed giving the name of the file which is locked.

Examples:

rm file1 file2
rm foo,s5

The first example deletes files "file1" and "file2" from the current data drive. The second example deletes the file "foo" from the disk in slot five. The drive number will be the same as the current data drive number.

2.6.14 run

run [argl] [arg2] ...

Does a jsr to the starting address of the last file loaded after pushing a pointer to the argument vector and the number of arguments on the stack. Argv[0] will be the "run" string.

Example:

load tabset
A=0800 L=12F0
run 8

This example loads the program "tabset" into memory. The SHELL displays the load address and length. The "run" command then calls hex 800 ($800) with the argument "8". The three lines are equivalent to typing:

tabset 8

all by itself.

2.6.15 save

save file [aN] [lN]

Saves a part of memory to a file on the specified device. If the starting address and length are not specified, the starting address and length of the last file "load"ed will be used if N is begun with a '$', the value is interpreted as a hex value otherwise as decimal.

Examples:

save foo
save foo a$800 l1000

The first example will save in a file called "foo", whatever the last program loaded or run. The second example will save a thousand bytes of memory starting at hex 800 in a file called "foo".

2.6.16 unlock

unlock file1 [file2] ...

Unlock the file on the specified slot, drive and volume. If any of slot, drive or volume are not given, they default to the current data values.

Examples:

unlock test1
unlock test1 test2 test3,d2

The first example unlocks file test1 on the current data disk. Example two unlocks files test1 and test2 on the current data disk and unlocks file test3 on drive two of the current data slot and volume.

2.7 Batch Facilities

Text files (7 bit sequential text files not DOS 3.3 text files with hi-bits set) containing a series of SHELL command lines can be executed by simply typing the name of the text file followed by a carriage return. Note that the type of the file must be 'T'. All SHELL

input is then taken from that file until the end is reached SHELL command files may not be nested, but they may be chained. If a SHELL command line executes a second SHELL command file, the first command file is closed and forgotten. Lines beginning with the '#' character are ignored by the shell and can be used as comments.

When the SHELL is booted for the first time, the disk that the SHELL was booted from is searched for a file called ".PROFILE". If this file is found and is a (7 bit not DOS 3.3) text file, it will be executed immediately. This allow any special startup procedures to be automatically initiated. SHELL command files may also be given up to 9 arguments. These arguments are referenced by the character '$' followed by the number of the argument to be used. Argument 0 is the name of the SHELL command file itself. For example, to link together several files, the following one line SHELL command file might be created:

ln -o In.out $1 $2 $3 $4 $5 $6 $7 $8 $9 sh65.lib

If the file was called "linkit", it could be used by typing:

linkit f1.rel f2.rel f3.rel

If an argument does not exist, it is ignored.

There are two special "built-in" commands that the SHELL will only recognize when read from a SHELL command file. These commands are used for additional control over the processing of the commands in a SHELL command file.

2.7.1 loop

loop

This command is used to start and end a loop in a SHELL command file. The command lines between the two loop statements will be executed once for each argument given to the SHELL command file. During the loop, two special arguments are available for use.

'$#' is replaced by the number of the current argument being processed. The two-character sequence '$%' will be replaced by the current argument itself. The following is an example of a SHELL command file which will compile and assemble from one to nine files, one at a time.

```
set -x -a
loop
# This is argument number $#, $%
c65 -a -o $%.asm $%.c
as65 -o $%.rel $%.asm
loop
```

If the preceding lines were placed in a file called "compile", then the statement:

compile test junk foo

would compile and assemble the three files "test.c", "junk.c", and "foo.c" into the corresponding ".rel" files and produce:

loop
This is argument 1, test
c65 -a -o test.asm test.c
as65 -o test.rel test.asm
loop
This is argument 2, junk
c65 -a -o junk.asm junk.c
as65 -o junk.rel junk.asm
loop
This is argument 3, foo
c65 -a -o foo.asm foo.c
as65 -o foo.rel foo.asm
loop

2.7.2 set

set [+-x] [+-a] [+-n]

Sets or clears one of three internal flags in the SHELL. Using '+' will clear the flag while '-' will set it. The flags are defined as follows:

x	**Echo command lines to the screen. Defaults to off.**
a	**Abort the SHELL command file if a command or program exits with a non-zero value. Defaults to no abort.**
n	**Parse the command lines but do not execute them. Defaults to off.**

Thus, to see each line being executed, the first line of a SHELL command file should be:

set -x

To have a SHELL command file exit if an error occurs, include the line:

set -a

The "set" command may only be executed within a SHELL command file.

2.8 Configuration

The basic Apple II is limited in its ability to deal with upper and lower case and has a limited screen size. The SHELL contains device drivers which allow it to overcome these limitations to some degree. However, these same routines have been set up to take advantage of optional peripherals which greatly enhance the Apple's operation. There are two approaches to dealing with peripheral devices, writing custom routines to deal with one particular device or to write a general routine to handle a number of similar devices. The original versions of the SHELL device drivers were examples of writing custom routines. The current version contains general purpose routines for dealing with three devices, the keyboard, screen and printer.

The device routines make use of a table at a fixed location in the driver to handle the functional differences between different hardware configurations. This table can be modified by using the CONFIG program provided on the STARTUP diskette. A separate set of options is available for each device and are detailed in the following.

2.8.1 Keyboard

The first device is the keyboard There are four variations of keyboard available. First, is a full upper and lower case keyboard as is available with the //e or a keyboard enhancer. If this option is selected, no mapping is done at all. Second, is an Apple keyboard with the single wire shift key mod installed, while the third is an Apple keyboard without the SWSKM. Both of these options map characters from the keyboard to get the full range of ASCII characters. Finally, it is possible to specify that the keyboard is a remote terminal. In this case, the driver will use the Pascal 1.0 entry point to the card that is assumed to be in slot 3. It will not do any mapping on the data received from the card. Also, since there is no status entry point, the AS and AC output control characters are not available. The AC abort is still enabled during input.

2.8.2 Screen

The second device is the screen. There are three types of screen. First, the basic Apple screen with 40 columns and upper case only. Second, 40 columns with upper and lower case capability. Examples of this are the //e and a II with a lower case adapter. Finally, there are the 80-column screens. All 80-column screens, remote and otherwise, are assumed to reside in slot 3 and are accessed by using the Pascal 1.0 output hook.

Not all 80-column screens are identical, and do not necessarily use the same control sequences to perform such functions as clearing the screen, moving the cursor and others. To minimize this problem, a table of control codes has been built into the SHELL device driver. This table is used by the ioctl() routine when performing the appropriate functions.

The values in this table are already known for several devices by the CONFIG program. The devices whose values are known are the //e, the Videx Videoterm, and the Smarterm. If a device is used which is not compatible with any of the above three cards, then the entries to the table must be provided by the user. The only programs which currently

make use of the ioctl() screen calls are the screen editor VED, and the CONFIG program. For VED, the only required functions are cursor positioning, clear to end of line, and clear screen. CONFIG only uses the clear screen.

2.8.3 Printer

The last device is the printer. The printer is assumed to be driven by a peripheral card in slot 1 with firmware which supports the PR# basic protocol. The printer is initialized by placing the address of the card in the CSW vector in low memory and then calling the card. Normally the card then replaces the address in the CSW vector with the normal character output routine. The printer driver then sends a string of characters to initialize the firmware on the card. The default sequence is:

^I^Y^Y255N

which tells the card that the width is 255 and not to echo the characters to the Apple screen. It is not really necessary to change the control character to be something other than a ^I since tabs are expanded to spaces by the print driver. After the driver sends the initialization string, it saves the address in CSW for use when sending characters to the printer. When sending characters, the address is placed back in CSW and control passed to the firmware by jumping indirectly through CSW.

The printer has three other control modes. First, some printer card firmware requires that the high bit be on for characters sent to it. If so, the driver has a flag which will cause it to "or" in a hex 80 with each character before transmitting it to the firmware. Also, the print driver automatically converts newlines (LF) to carriage returns before transmitting to the firmware on the card. If the appropriate flag is set, the print driver will automatically send a line feed after each carriage return. Finally, when the printer is closed it is possible to have the print driver automatically send a form feed ($0C) character to the device. All these flags are set by answering the appropriate questions in the CONFIG program.

PROGRAMS

3. PROGRAMS

3.1 C65 Native Code Compiler

c65 [-abts] [-o file] [-Dtoken] [-Enn]
[-Xnn] [-Ynn] [-Znn] file.c

The Aztec C65 compiler is a true native code C compiler. C65 produces in-line assembly language code for all C statements with the following exceptions:

*** All floating point operations.**
*** Multiplication, division, and modulus.**
*** Shifts.**
*** All pseudo-stack operations.**
*** Switches.**
*** Structure copies.**

The code generated by the compiler uses a 16 bit pseudo- stack pointer kept in locations 2-3 of zero page. This stack is used for all local variable storage and for passing arguments to functions. The return address of function calls is also stored on the pseudo-stack. The 6502 machine stack is only used for temporary storage, thus fully recursive programming may be used without the limitations of the 6502 machine stack.

C65 makes use of the first thirty-two locations of zero page as work space and temporary registers. C65 also uses locations $80-$8F of zero page as user declared register variables. Up to eight "register" declarations are accepted within each function. Each routine which uses register variables automatically saves the locations it uses on the pseudo-stack and restores them when it exits. Chars, ints, unsigned ints and pointers may be declared as registers.

Use of register variables produces significantly smaller and faster code. The hidden overhead of saving and restoring register variables is minor compared to the gain in speed and code size. The simplest use of the compiler is just

c65 name.c

It is recommended that the file name end in ".c", but it is not necessary. C source statements found in the "name.c" file are translated to 6502 assembler source statements and written to a file named "$TMP.$$$". Then the compiler will automatically execute the AS65 assembler which will assemble the "$TMP.$$$" file and produce a relocatable object file called "name.rel". The "$TMP.$$$" file will then be deleted by the assembler.

The options available with C65 are listed below.

-a

If it is necessary to view the assembly language produced by C65, this option will force the compiler to leave the output in a file called "name.asm", where "name" is from the first part of the file being compiled. In this instance, the assembler will not be executed. For example:

c65 -a dbms.c

will leave the assembly language output in the file "dbms.asm".

-o

This option allows the user to specify the name of the output file. Normally, this can be used to specify the name of the relocatable object module as in:

c65 -o temp.rel dbms.c

which compiles "dbms.c" and then assembles the "$TMP.$$$" file and places the output of the assembler in "temp.rel".

When used with the "-a" option, it specifies the name of the assembly language file instead, as in:

c65 -a -o temp.asm dbms.c

which compiles "dbms.c" and places the assembly language in the file "temp.asm" and quits.

-b

Normally, when conditionals are evaluated, the compiler generates a test and a branch around a "jmp" instruction since it cannot know that the branch will be in range. For example:

cmp #45
beq .5
jmp .17

This option will force the compiler to generate a direct branch instead of the branch and jump, as in:

cmp #45
bne .17

If the branch is too long, an error message will not be generated until the module is linked Most of the library was compiled with this option.

-s

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.

-t

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

-D

This option allows a token to be entered into the macro definition table as being defined This is most useful for controlling the conditional compilation of code. For example, if a section of code is to be included for a specific customer, it might be surrounded by an ifdef-endif combination:

```
#ifdef CUSTOM
statement1;
statement2;
statement3;
#endif
```

When normally compiled, these statements would not be included, but when compiled with:

c65 -DCUSTOM prog.c

the statements would be compiled into the program. Multiple uses of the "-D" option are permitted on one command line. There are four options for changing default internal table sizes.

-E

The "-E" option specifies the size of the expression work 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:

c65 -E300 prog.c

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

-X

The "-X" option specifies the size of the macro (#define) work table. 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:

c65 -X4000 prog.c

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

-Y

The "- Y" option specifies the maximum number of outstanding cases allowed in a switch statement. 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;
case 1:
break;
case I:
  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:

c65 -Y300 prog.c

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

-Z

The "-Z' option specifies the size of the string literal table. 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:

c65 -Z3000 prog.c

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.

The AZTEC C native-code compiler is implemented according to the language description supplied by Brian W. Kernighan and Dennis M. Ritchie in The C Programming Language (1st Edition). The user should refer to that document for a

description and definition of the C language. This document has detailed 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 reinforce 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.

Information regarding interfacing with assembly language can be found in the Technical Information section of this manual

Note: The C Programming Language (1st Edition)

The version of C described in The C Programming Language (1st Edition) published in 1978 is referred to as K&R C, to distinguish it from ANSI C. In 1988 ANSI C was first standardized and The C Programming Language (2nd Edition) was published to cover ANSI C.

Aztec CII Version 1.05i 6502 (this compiler) was released in 1983 (the same year that ProDOS 8 was first released). It is exclusively for DOS 3.3 and uses K&R C. By the time ANSI C came along, the Aztec CII compiler for the Apple II had been replaced by Aztec C65 Version 3.2b (June 1987) which offered support for both DOS 3.3 and ProDOS (which had taken over from DOS 3.3 by that time). The newer Version 3.2b also provided support for creating programs for the newer Aztec C ProDOS Shell, but dropped support for the DOS 3.3 Aztec C Shell (this shell).

Aztec C65 Version 3.2b provided many ANSI C features but was still a K&R C compiler. Version 3.2b was the last Aztec C compiler released for the Apple II. It had already been released for a year (June 1987) by the time ANSI C first came along and The C Programming Language (2nd Edition) was published. An Aztec C65 ANSI C compiler for the Apple II was never released.

3.2 CCI Pseudo-code Compiler

cci [-ats] [-o file] [-Dtoken] [-Enn]
[-Xnn] [-Ynn] [-Znn] file.c

The Aztec CCI compiler is a pseudo-code compiler. CCI produces assembly language for a pseudo- machine that is interpreted by a 6502 assembly language program. The pseudo-machine makes use of the same pseudo-stack and registers as the native code produced by C65. Thus, there is no difficulty in mixing routines compiled using C65 with routines compiled with CCI.

CCI differs from C65 in that it does not make use of the "register" type definition. The declaration is allowed; but, it is simply ignored.

CCI is invoked by the command:

cci name.c

It is recommended that the file name end in ".c", but it is not necessary. C source statements found in the "name.c" file are translated to pseudo-code assembler source statements and written to a file named "$TMP.$$$". Then the compiler will automatically execute the ASI assembler which will assemble the "$TMP.$$$" file and produce a relocatable object file called "name.int". The "$TMP.$$$" file will then be deleted by the assembler.

The options available with CCI are the same as those for C65, with the exception of the "-b" option, which does not apply to the pseudo-code compiler. Please refer to the section on C65 for more information.

Note: The last comment seems to be false and may have been left in from the manual of an earlier version of CCI. This version of C65 does in fact allow base address selection for Aztec CCI compiled programs for the DOS 3.3 Shell.

3.3 AS65 6502 Assembler

as65 [-c] [-l] [-ZAP] [-o file] file.a65

3.3.1 Overview

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

as65 test.a65

The file "test.a65" is the assembly language source file. The filename does not have to end in ".a65". In this case, the relocatable object file produced by the assembler will be named "test.rel" where test is the same name as the prefix of the input filename. The type of the file in a CATALOG will be 'R'. There are several options to the assembler which are detailed below.

-o

An alternative object filename can be supplied by specifying the option "-o filename". The object file will be written to the filename following the "-o". The filename does not have to end in ".rel". It is, however, the recommended format.

-c

This option forces the assembler to make two passes through the source file. This allows most forward references to be resolved during the second pass. The overall result is that the object file size is significantly smaller since very few local labels need to be stored in the object module. This option was added primarily for the production of libraries, where size of the module is important. The overhead of reading the source file twice makes this option much less useful during normal compilation and assembly with one exception. If the "-b" option of C65 is used, using the "-c" option will detect a branch out of range without having to use the linker.

-l (lower case "L")

This option generates a listing of the assembly language file. All opcodes are specified in the listing and all arguments that are known. Unknown arguments such as forward branches and addresses are represented as "XX". Using the "-c" and the "-l" together eliminates the "XX"'s in forward branches. The output is placed in a file with a ".lst" extension.

-ZAP

This option forces the assembler to delete the input file after performing the translation. This option is used by C65 when it automatically executes the assembler to delete the temporary file "$TMP.$$$".

3.3.2 Syntax

The following defines the syntax for the AS65 assembler.

Statements

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

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

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

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.

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
```

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.

Assembler Directive

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**

3.4 ASI Pseudo-code Assembler

asi [-ZAP] [-o file] file.asm

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

asi name.asm

The relocatable object file produced by the assembly will be named "name.int" where name is the same name as the name prefix on the ".asm" file. The type of the file in a CATALOG of the disk will be 'R'. An alternative object filename can be supplied by specifying "-o filename". The object file will be written to the filename following "-o". The filename does not have to end with ".int"; it is, however, the recommended format. The file "name.asm" is the pseudo-code assembly language source file. The filename does not have to end in ".asm".

The "-ZAP" option forces the assembler to delete the input file after performing the translation. This option is used by CCI when it automatically executes the assembler to delete the temporary file, "$TMP.$$$".

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

3.5 LN Linker

ln [-t] [-o outfil] [-r] [-b N] [-c N]
[-d N] [-f infil] file.rel ...

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 utility and produce an executable binary file.

Supplied with the AZTEC C Compiler System are several object libraries. In most cases one or more of these libraries must be specified. To link a simple single module routine, the following command will suffice:

ln name.rel libname.lib

The operand "name.rel" is the name of the object file. The executable file created by LN will be named "name". 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.rel mod2.int mod3.rel libname.lib

Also several libraries can be searched as in the following:

ln -o name mod1.rel mod2.int mine.lib libname.lib

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.rel mine.lib mine.lib libname.lib

The link editor will read the "mine.lib" 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 ".sym".

-b address

to specify a base address other than hex 800. The base address is normally the lesser of the code start address and the data start address, but may be lower than either. The "base address" is assumed to be in hex.

-c address

to specify a starting address for the 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 immediately at the end of the code segment. The three preceding options are usually used when producing ROMable code or for similar reasons. More information on ROMable code can be found in the Technical Information section of this manual. These options were used to link the SHELL so that the data was located outside the language card. This allowed the language card to be write protected. The command to link the SHELL was basically:

ln -o SHELL -b A7FD -d A800 -c D000 -f shell.lnk

The base address was set at three below the data since the linker automatically places a "jmp" to the start of the code at the base address unless the base and the code address are the same.

-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.

-r

This option is used to inform the linker that the modules being linked are the root segment of a program with overlays. With this option, a file with a ".rsm" extension will be produced which is used in linking the overlays. More information on overlays can be found in the Technical Information section of this manual.

3.6 MKLIB

mklib [-atxr] [-o library] [.] module1 module2 ....

This program creates libraries which can be used by the linker, LN, in a very efficient manner. Each module is individually rearranged to make the linking process as fast as possible. In addition, a dictionary of global variables which are in the modules is automatically created as part of the library. This dictionary is used by the linker so that it only looks at those modules that it needs to.

Note that a library may be specified as a module. In that case, all the modules in the library are copied into the new library. This can be used to combine several libraries into a single library.

There are several options to MKLIB which are detailed below. The simplest use, however is to create a new library, as in:

mklib mod1.rel mod2.rel mod3.rel

which creates a library called "libc.lib" with the three named modules. As a convenience, if the number of modules to be added to a library are large, the "." option can be used. When the program processes the "." in the argument list, it switches its argument parsing to the standard input. Thus, if a file containing the names of all the modules to be linked has been created called "infil", then

mklib . < infil

will create a library called "libc.lib" with the modules named in "infil".

-o library

This option specifies the name of the library to be created or to be modified. The default name of "libc.lib" is used if this option is not specified. For example:

mklib -o mylib.lib file1.rel file2.int . < infil

places the output in "mylib.lib".

-t

This option lists the names of the modules in the library. Note that a module may contain several functions and that the name of the module may have nothing to do with the names of the functions within that module. Module names are derived from the names of the files used to create the library. As an example:

mklib -t -o mylib.lib

will list the module names of "mylib.lib".

-r

This option copies the library module by module. If a module name matches the name of one of the modules specified as an argument, the module is not copied from the library, but from the object file instead. This process continues until the end of the library is reached. At that point, the remaining module names are appended to the library. The original library is deleted, and the new library renamed.

-a

This option appends the named modules to the end of the library. All modules specified will be appended. However, in order to update the dictionary properly, the library will be copied in the process.

-x

This option extracts the named library modules from the library into individual files. If no module names are specified, all modules are extracted.

3.7 VED Screen Editor

ved [-tn] [file]

VED is a screen oriented text editor written in C for use with the Aztec C65 system. The source to VED is included in the archive "VEDSRC.ARC". VED is not a particularly fast or smart editor, but it does get the job done. If VED is invoked with a file name, that file will be loaded into the memory buffer, otherwise it will be empty. VED does all its editing in memory and is thus limited in the size of files that it will edit. In VED, the memory buffer is never completely empty. There will always be at least one newline in the buffer.

The "-t" option specifies that a different tab size should be used. Normally VED will use the current system value, but this may be overridden with this option, as in:

ved -t8 file.a65

which is useful since C programs work well with a tab size of four, but assembly language works better with a tab size of eight.

VED has a 1000 character limit on the size of a line. If a line is longer than the width of the screen, it will wrap to the next line. If a line starts at the bottom of the screen, and is too wide to fit, the line will not be displayed Instead, the '@' character will be displayed. Likewise, at the end of the file, all lines beyond the end will consist only of a single '-' on each line.

A number of commands take a numeric prefix. This prefix is echoed on the status line as it is typed.

The normal mode of VED is command mode. During command mode, there are a number of ways to move the cursor around the screen and around the whole file.

newline	**move to the beginning of the next line.**
-	**move to the start of the previous line.**
space	**move to the next character of the line.**
backspace	**move to the previous character.**
0	**move to the first character of this line.**
$	**move to the last character of this line.**
h	**move to the top line of the screen.**
l	**move to the bottom line of the screen.**
b	**move to the first line of the file.**
g	**move to the n'th line of the file.**
/string	**move to the next occurrence of 'string'.**

When the cursor is in the appropriate spot, there are two commands used to delete existing text.

x	**delete the n character under the cursor up to but not including the newline.**
dd	**delete n lines starting with the current line.**

Note that deleting the last character on the line (newline character) causes the following line to be appended to the current line.

To add new text, hitting the 'i' key will cause the top line of the screen to indicate that you are now in <INSERT> mode. To exit insert mode, type ESCAPE (unless the CAPS LOCK mode is enabled, in which case type control-Q). To insert a control character which means something special to VED into a text file, first type control-V followed by the control character itself. Control characters are displayed as '^X', where X is the appropriate character.

Typing 'o' will cause a new line to be created below the current line, and the cursor will be placed on that line and the editor placed into <INSERT> mode.

There are three commands used for moving text around. These commands make use of a 1000 character yank buffer. The contents of this buffer is retained across files.

yy	**yank n lines starting with the current line into the yank buffer.**
yd	**yank n lines starting with the current line and then delete them.**
p	**"put" the lines in the yank buffer after the current line.** **The yank buffer is not modified.**

The 'z' command redraws the screen with the current line in the center of the screen. The 'r' command replaces the character under the cursor with the next character typed.

When in command mode, if the ':' key is hit, a ':' will be displayed on the status line. At this point, a number of special file-related commands may be given.

:f	**displays info about the current file.**
:w file	**writes the buffer to the specified file name.**
:w	**writes the buffer to the last specified file.**
:e file	**clears the buffer and reads the named file.**
:e! file	**clears the buffer and reads the named file even if the file was modified**
:r file	**reads the named file into the buffer.**
:q	**exits the editor.**
:q!	**exits editor even if the file was modified**

As can be seen VED protects from accidentally destroying the work being edited by preventing exiting or editing another file if the current file has been modified. If the file has been written using the ":w" command, the modified flag will be cleared.

VED will only edit Apple text files. Binary files will not be edited.

3.8 ARCH Source Archive Utility

arch -[clvxa]o archive [-f infil] [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. The 'o' option must always be used to specify the name of the archive itself. Only one of the options 'lxa' may be specified. The 'v' option is a modifier for the 'xa' options and causes them to print each file name they act upon. The remaining options are detailed below.

-l

This option lists the named files in the archive, giving the name and size of each. If no file names are specified, all of the files in the archive are listed For example:

arch -lo progsrc.arc

lists the names and sizes of the files in the "progsrc" archive.

-a

This option specifies that the named files be appended to the end of the archive. If the 'c' option is given as well, the archive is truncated before adding the files.

-x

This option extracts the named files from the archive. If no file names are given, all the files in the archive are extracted from the archive. The archive is not modified.

-f file

This option forces the ARCH program to read the named "file" for the names of the files to be placed in the archive. All the lines in the file are read, and more than one file name may be placed on each line.

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

3.9 OD Hex Dump Utility

od [+nnn[.]] 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. If the optional argument "+nnn" is supplied. "nnn" is used as an offset into the file where the dumping is to start. If "nnn" is followed by a '.', it is treated as a decimal number, otherwise it is considered to be a hex value. Each file will be dumped starting from the last offset argument encountered. Thus, an offset of "+0" will cause the files which follow it to be dumped from their beginning.

For example:

od + 16b oldtest newtest +0 junk
od + 1000. tstfil

Note: The OD program is a hex viewer with an 8 byte display width. The OD source code is provided with Aztec C. The RD program written in 2013 used the OD program as a starting point and modified it for the 80 column display and added many features. RD is distributed with source code as part of the Aztec33 distribution.

3.10 CMP Byte for Byte File Compare

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. The program will normally stop after the first difference, unless the "-l" option is given. If the "-l" option is specified. CMP will list all differences in the format:

decimal offset	hex offset	file1 value	file2 value

If no difference is found, the program will exit without saying anything.

For example:

cmp -l otst ntst

10 a: 00 45
100 64: 1a 23

and

cmp otst ntst

Files differ character 10.

3.11 NM Name List Generator

nm [-sunago] file1.int [file2.rel] ...

This utility operates only on the relocatable object files which are the output of the two assemblers, AS65 and ASI. 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 valuc 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:

-s	**Display only the size of the code and data.**
	-or-
-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.**

For example, to see the size of several modules:

nm -s mod1.rel mod2.int mod3.rel

or to see the undefined global symbols sorted in alphabetic order:

nm -uga mod1.rel mod2.int mod3.rel

3.12 TABSET

tabset [newsize]

This program displays the current setting of the tab width parameter of the SHELL. If the argument is specified, the tab width parameter is set to that value. In that case, both the old and the new value are displayed. The parameter is stored in location $D088 of bank 1 of the RAM card.

3.13 CONFIG

config

This program takes no parameters as it is completely interrogative. CONFIG is used to alter the SHELL's device driver tables and thus make use of any non-standard peripherals. More information on the use of the CONFIG program can be found in the SHELL section of this manual.

3.14 LDEV

ldev file

This program replaces a former "built-in" SHELL command. LDEV loads the named file into bank 1 of the RAM card and is used when loading a new or custom set of device drivers. The format of the device driver module is the same as that previously used. However, to take advantage of new features such as the settable tab width parameter, custom drivers will need to be incorporated into the new SHELL drivers. More information is available in the SHELL section and in the Technical Information section of this manual.

LIBRARIES

4. LIBRARIES

4.1 Introduction

The libraries provided with the Aztec C65 system can be divided into four logical groupings. These groups are the standard I/O, system I/O, utility, and math/floating point libraries. The source to all the libraries with the exception of the math library is provided with the system as archives.

The compiled object modules are supplied in three libraries. However, six library files are supplied with the system, three are compiled with C65 and three are compiled with CCI.

The first library is the floating point library. This library contains the floating point emulation routines and all of the transcendental math functions. This library must be specified if any floating point operations are performed in any module being linked. If the library is not specified, the linker will abort with the symbol ".fltused" undefined This library must be specified before the regular library for successful operation.

The names of this library are

FLT65.LIB and FLTINT.LIB

which correspond to the C65 compiled version and the CCI compiled version.

The remaining two libraries are similar in function. The primary difference between the libraries involves their use of the SHELL. Since the SHELL contains many of the system I/O routines and a number of the utility routines, including the pseudo-code interpreter, these routines are not included in one of the libraries. Instead, a set of dummy addresses is included which provide a link to the routines within the SHELL.

Programs which use the SHELL vector are smaller and therefore take less disk space and load faster. These programs also make use of the configured SHELL device drivers.

The names of the SHELL libraries are

SH65.LIB and SHINT.LIB.

The non-SHELL library contains all the routines in the SHELL library as well as all the routines used by the SHELL. This library is known as the stand-alone library, since programs linked with this library can be run without the SHELL in a normal DOS environment. The one significant difference, other than size, of programs linked with this library is that of console I/O. The I/O drivers supplied with the stand-alone library are not the same as those contained in the SHELL. The calling format and use is the same, but

the actual routines are much simpler. More information on this subject can be found in the Technical Information section of this manual.

The names of the stand- alone libraries are

SA65.LIB and SAINT.LIB

The differences between the libraries compiled with C65 and CCI are minimal, mostly relating to size and speed. Any program may be linked with either library without any hesitation or special procedures.

4.2 Summary of Library Functions

4.2.1 Standard I/O

agetc	**(stream)**	**ASCII version of getc**
aputc	**(c,stream)**	**ASCII version of putc**
clearerr	**(stream)**	**clears the error flag on stream**
exit	**(return)**	**flushes and closes all streams**
fclose	**(stream)**	**closes an I/O stream**
feof	**(stream)**	**check for eof on stream**
ferror	**(stream)**	**check for error on stream**
mush	**(stream)**	**write out buffered data to stream**
fgetc	**(stream)**	**gets a single character from stream**
fgets	**(buffer, max, stream)**	**reads line from stream to buffer**
fileno	**(stream)**	**returns the fd associated with stream**
fopen	**(name, how)**	**opens file name according to how**
fprintf	**(stream, format, arg1, ...)**	**formatted print to stream**
fputc	**(c, stream)**	**writes character c to stream**
fputs	**(cp, stream)**	**writes string cp to stream**
fread	**(buf, sz, cnt, stream)**	**reads cnt items from stream to buf**
freopen	**(name, mode, stream)**	**switches stream to new file**
fscanf	**(stream, cntrl, p1, ...)**	**converts input string from stream**
fseek	**(stream, pos, mode)**	**positions stream to pos**
ftell	**(stream)**	**returns current file position**
fwrite	**(buf, sz, cnt, stream)**	**writes cnt items from buf to stream**
getc	**(stream)**	**gets a single character from stream**
getchar	**()**	**gets a single character from stdin**
gets	**(buffer)**	**reads a line from stdin**
getw	**(stream)**	**gets a word from stream**
printf	**(format, arg1, ...)**	**writes formatted data to stdout**
putc	**(c, stream)**	**writes character c to stream**
putchar	**(c)**	**writes character c to stdout**
puts	**(cp)**	**writes string cp to stdout**
putw	**(w, stream)**	**writes a word w to stream**
rewind	**(stream)**	**position stream at beginning**
setbuf	**(stream, ut)**	**force stream to use buf**
scanf	**(cntrl, p1, ...)**	**converts input string from stdin**
sprintf	**(cp, format, arg1, ...)**	**formats data into string cp**
sscanf	**(cp, cntrl, p1, ...)**	**converts input string cp**
ungetc	**(c, stream)**	**pushes c back into stream**

4.2.2 System I/O

exit	**(return)**	**returns control to operating system**
catalog	**(slot, drive, volume)**	**do a "CATALOG" of the disk**
chmod	**(name, how)**	**lock or unlock file name**
close	**(fd)**	**closes file fd**
creat	**(name, mode)**	**creates a file of type mode**
ioctl	**(fd, cmd, arg)**	**perform special I/O function**
lseek	**(fd, pos, mode)**	**positions file fd according to mode**
open	**(name, rwmode)**	**opens file according to rwmode**
read	**(fd, buf, size)**	**reads size bytes from file fd to buf**
rename	**(oldname, newname)**	**renames a disk file**
unlink	**(filename)**	**deletes a disk file**
write	**(fd, buf, size)**	**writes size bytes from buf to file fd**

4.2.3 Utility Routines

alloc	**(size)**	**allocates size bytes**
atof	**(cp)**	**converts ASCII to floating**
atoi	**(cp)**	**converts ASCII to integer**
atol	**(cp)**	**converts ASCII to long**
blockmv	**(dest, src, size)**	**moves size bytes from src to dest**
calloc	**(nelem, elsize)**	**allocates space for nelem*elsize**
clear	**(area, size, value)**	**initialize area to value**
execl	**(prog, arg1, arg2, ...)**	**executes prog with args**
format	**(func, format, argptr)**	**outputs formatted data using func()**
free	**(addr)**	**frees the space at addr**
ftoa	**(m, cp, prec, type)**	**converts floating to ASCII**
htoi	**(cp)**	**converts ASCII hex to integer**
index	**(cp, c)**	**finds c in string cp**
isdigit	**(c)**	**checks for digit 0...9**
islower	**(c)**	**checks for lower case a...z**
isspace	**(c)**	**checks for white space**
isupper	**(c)**	**checks for upper case A...Z**
malloc	**(size)**	**allocates size bytes**
rindex	**(cp, c)**	**finds c in string cp backwards**
rwts	**(tr,se,buf,cmd,sl,dr,vol)**	**read or write a sector from disk**
settop	**(size)**	**bumps top of program memory by size**
stcrat	**(str1, str2)**	**appends string 2 to the end of string 1**
strcmp	**(str1, str2)**	**compares string 1 with string 2**
strcpy	**(str1, str2)**	**copies string 2 to string 1**
strlen	**(str)**	**returns length of string**
strncat	**(str1, str2, n)**	**appends at most n character**
strncmp	**(str1, str2, n)**	**compares at most n characters**
strncpy	**(str1, str2, n)**	**copies at most n characters**

system	**(str)**	**SHELL executes string str**
tolower	**(c)**	**converts to lower case**
toupper	**(c)**	**converts to upper case**

4.2.4 Math Routines

acos	**(x)**	**inverse cosine of x (arcos x)**
asin	**(x)**	**inverse sine of x(arcsin x)**
atan	**(x)**	**inverse tangent of x (arctan x)**
atan2	**(x, y)**	**arctangent of x divided by y**
cos	**(x)**	**cosine of x**
cosh	**(x)**	**hyperbolic cosine of x**
cotan	**(x)**	**cotangent of x**
exp	**(x)**	**exponential function of x**
log	**(x)**	**Natural log of x**
log10	**(x)**	**logarithm base 10 of x**
pow	**(x, y)**	**raise x to the y'th power**
ran	**()**	**random number from 0 to 1**
sin	**(x)**	**sine of x**
sinh	**(x)**	**hyperbolic sine of x**
sqrt	**(x)**	**square root of x**
tan	**(x)**	**tangent of x**
tanh	**(x)**	**hyperbolic tangent of x**

APPENDICES

APPENDIX A: Compiler Error Codes

1	bad digit in octal constant
2	string space exhausted
3	unterminated string
4	internal error
5	illegal type for function
6	inappropriate arguments
7	bad declaration syntax
8	syntax error in typecast
9	array dimension must be constant
10	array size must be positive integer
11	data type too complex
12	illegal pointer reference
13	unimplemented type
14	internal
15	internal
16	data type conflict
17	unsupported data type
18	data type conflict
19	obsolete
20	structure redeclaration
21	missing }
22	syntax error in structure declaration
23	obsolete
24	need right parenthesis or comma in arg list
25	structure member name expected here
26	must be structure/union member
27	illegal typecast
28	incompatible structures
29	illegal use of structure
30	missing : in ? conditional expression
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
37	invalid expression for unary operator
38	no auto. aggregate initialization allowed
39	obsolete
40	internal
41	initializer not a constant
42	too many initializers

43	initialization of undefined structure
44	obsolete
45	bad declaration syntax
46	missing closing brace
47	open failure on include file
48	illegal symbol name
49	multiply defined symbol
50	missing bracket
51	l value required
52	obsolete
53	multiply defined label
54	too many labels
55	missing quote
56	missing apostrophe
57	line too long
58	illegal # encountered
59	macro too long
60	obsolete
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	goto needs a label
71	statement syntax error in do-while
72	'for' syntax error; missing first semicolon
73	'for' syntax error;; missing second semicolon
74	case value must integer constant
75	missing colon on case
76	too many cases in switch
77	case outside of switch
78	missing colon on default
79	duplicate default
80	default outside of switch
81	break/continue error
82	illegal character
83	too many nested includes
84	too many array dimensions
85	not an argument
86	null dimension in array
87	invalid character constant

88	not a structure
89	invalid use of register storage class
90	symbol redeclared
91	illegal use of floating point type
92	illegal type conversion
93	illegal expression type for switch
94	invalid identifier in macro definition
95	macro needs argument list
96	missing argument to macro
97	obsolete
98	not enough arguments in macro reference
99	internal
l00	internal
101	missing close parenthesis on macro reference
102	macro arguments too long
103	#else with no #if
104	#endif with no #if
105	#endasm with no #asm
106	#asm within #asm block
107	missing #endif
108	missing #endasm
109	#if value must be integer constant
110	invalid use of : operator
111	invalid use of void expression
112	invalid use function pointer
113	duplicate case in switch
114	macro redefined
115	keyword redefined

Error codes greater than 200 shouldn't occur. If they do, there's something wrong with the compiler. If you get such an error, please send us the program that generated the error.

APPENDIX B: The Graphics Disk

B.1 FILES ON THE GRAPHICS DISK (DISK7)

There 8 files on the graphics disk. The list below gives a brief synopsis of each:

g.lib	**The object library containing the graphics routines.**
g.arc	**The archive file containing the source for the graphics library.**
democlr.c and dem2clr.c	**Two demo programs using g.lib; they are written for color monitors.**
demoblk.c and dem02blk.c	**Two demo programs using g.lib; they are written for a black and white monitor.**
cpdemoclr	**A file with the correct compile, assemble and link steps for the two color demo programs.**
cpdemoblk	**A file with the correct compile, assemble and link steps for the two black and white demo programs.**

The only difference between the the color and black and white demo programs is one uses color and the other does not. If you have a color monitor use the color demo programs, if you have a black and white monitor use the black and white demo programs.

B.2 GRAPHIC FUNCTIONS

The graphics library g.lib contains graphic functions for the Apple. These functions allow programs to plot points, draw lines, circles, clear the screen. These functions use the high resolution graphics page, and the text pages. Programs can access both pages without any loss of data.

Note: This is a very detailed area for discussion, but to summarize:

In this old compiler for DOS 3.3 it was not possible to create a memory hole in a graphics program to load it over the HIRES screen. DOS 3.3 programs use HGR page one which starts at 0x2000 and ends just before 0x4000, so a DOS 3.3 program that uses HGR must run either below HGR or above HGR.

If you are writing graphics programs for the shell it is possible to use the memory below HGR and link to g.lib without a "special" base address being specified for the linker because shell programs are smaller. If you writing "RAW" DOS 3.3 programs that use HGR they must generally have their base address set to 0x4000 above HGR using

LN –b 4000 my.rel g.lib flt65.lib sa65.lib

In the newer compiler version 3.2b you can create a memory hole over HGR and your "RAW" DOS 3.3 or ProDOS shell programs will run below and above the memory hole

leaving the screen alone so you can write larger graphics programs without breaking them into overlays. But version 3.2b introduced a new problem because it supported ProDOS and SYS programs (ProDOS StandAlone programs) start at 0x2000 right at HGR. So ProDOS SYS programs need to use HGR2 (page 2) graphics (which is why I extended g.lib for ProDOS in that version and call it G2.lib. I also extended the DOS 3.3 g.lib for that version which still uses HGR and not HGR2. I call it g3.lib.

When I extended these libraries, I added routines for LGR, DLGR, and DHGR as well for DOS 3.3, ProDOS, and the ProDOS Shell. I have not done so for this version, but I have some extensions for LGR and HGR for the DOS 3.3 Shell in Aztec33.

For "RAW" DOS 3.3 programs that use graphics, a better alternative to this version is the DOS 3.3 3.2b version which is more robust than this old compiler but cannot exit and re-enter BASIC without a reboot. The 3.2b compiler is separated into 2 parts; AppleX which is the ProDOS and ProDOS Shell compiler, and Apple33, the "RAW" DOS 3.3 compiler; same compiler, different libraries. This compiler however was the last compiler for the DOS 3.3 shell's tiny programs and also contains a routine called rwts() Read Write Track Sector which is not available in the 3.2b compiler.

You will need to decide for yourself if you want to use this compiler or the newer one to do "RAW" DOS 3.3 graphics programs (or for that matter any "RAW" program). All of this runs under XP and before, and runs in Vista under DOSBox. Runs on Ubuntu under DOSEmu presumably too. And probably under DOSBox in other linuses and unises.

And one last thing; With Windows 7, this compiler quit running in DOSBox; it goes into a continuous loop and apparently can't seek source files so keeps repeating. VirtualBOX with FreeDOS is reported to work under Windows 7 but drive-sharing is not available due to a crippled installer since VirtualBOX apparently has strong linux support but no longer cares to provide a working Windows installer that I and others have been able to make work. If it works it must be designed for a non-windows user of some sort, because after developing software and working MS-DOS, Unix, Windows, and Linux for up to and over 30 years, I can't make it work, nor apparently can others since an ftp server is the only file transfer option that seems to work, and that is a worse cludge than some of this stuff that was written in 1982.

So the message here is that you may have only two options. The later version 3.2b DOS 3.3 compiler works in DOSBox under Windows 7 and will build raw DOS 3.3 programs. The Native Mode compiler that this document is about works almost as advertised. Since it runs in an Apple II Emulator everywhere Apple II emulators run, and since Apple II emulators run pretty quickly if they have a fast speed (AppleWin, kegs32) your only limitation will be switching DOS 3.3 floppy disks between compiling and linking. Aztec33 provides samples of doing this in its LS33.dsk and DIR33.dsk disk images which also are bundled with this compiler.

B.2.1 Overview

All the Hi-resolution (hires) graphics routines that plot points have two things in common:

1. **They all use 1 dot in the 280 by 192 matrix as 1 (x, y) location.**
2. **The upper left hand corner of the screen is considered location (0, 0).**

B.2.2 plotchar

plotchar -prints a character on the screen while in Graphics mode

plotchar (num, x, y)
char num, x, y;

plotchar will print any printable ASCII character on the screen at location x and y while in the Hi-Resolution mode. The character set is defined in _chr[] in the file 'graphvar.h'. plotchar is expecting to receive the integer value of the ASCII character. This routine does not check to see if the arguments are out of range.

Note: Other distributions of the newer 3.2b compiler have an extended character set that is available without this nonsense and can be loaded from a file as well into unused memory below your program. These routines can be adapted to this compiler.

B.2.3 circle

circle -draws a circle on screen

circle (x, y, rad)
int x, y, rad;

set_asp(x_asp, y_asp)
int x_asp, y_asp;

circle draws a circle on the screen with a center point of (x, y) with a radius of rad. The circle routine does not check to see if the circle is within range of the Hi-resolution page.

set_asp allows you to alter the shape of the circle so that it becomes an oval. Where x_asp and y_asp are equal to 1 the circle will be round. In any given case the circle will be round if x_asp = y_asp. If x_asp and y_asp are equal the circle will be plotted with a radius of (x_asp * radius) and (y_asp * radius) giving a circle that is round. If the values are > 1 the circle will be larger than its radius and if the values are < 1 the circle will be smaller than its radius. x_asp and y_asp are defined in graphvar.h and are initialized to 1.

Note: The circle routine described above is the only reason that the bloated floating point library needs to be linked to. Aztec33 has an integer based replacement that

eliminates this dependency which has been backported as an include file and eliminates the need to link to FLT65.lib unless you also need to use floats[4 bytes] and doubles[8 bytes] which will slow your programs down considerably.

B.2.4 line routines

line routines -draw white, blue, green, red, violet lines

drw (x1, y1, x2, y2)
int x1, y1, x2, y2;

bdrw (x1, y1, x2, y2)
int x1, y1, x2, y2;

gdrw (x1, y1, x2, y2)
int x1, y1, x2, y2;

rdrw (x1, y1, x2, y2)
int x1, y1, x2, y2;

vdrw (x1, y1, x2, y2)
int x1, y1, x2, y2;

lineto (x, y)
int x, y;

blineto (x, y)
int x, y;

glineto (x, y)
int x, y;

rlineto (x, y)
int x, y;

vlineto (x, y)
int x, y;

The drw routines plot a line from the ordered pairs (x1, y1) to (x2, y2). These routines do not check to see if any of the ordered pairs are out of range. These routines will plot a line in color depending on which routine is called. The color that goes with each drw each routine is shown below. The drw routines reset the the gobal variables _oldx, _oldy to the values of (x2, y2).

drw plots a white line
bdrw plots a blue line

gdrw plots a green line
rdrw plots a red line it
vdrw plots a violet line

The lineto routines plot a line from _oldx, _oldy to the ordered pair (x, y). The golbal variables _oldx _oldy are defined in the file 'graphvar.h' and are set to (0, 0). If no drw routines are called before a lineto routine the line will start at (0, 0). The color that goes with each lineto routine is shown below.

lineto plots a white line
blineto plots a blue line
glineto plots a green line
rlineto plots a red line
vlineto plots a violet line

Note: Version 3.2b has more robust revisions of this whole lot. I have backported some stuff from that version but left this part alone.

B.2.5 page

page -select which page is the current page

pagel ()
page2 ()

- pagel will set set whichever mode (Text or Hi-resolution) to the primary page.
- page2 will set set whichever mode (Text or Hi-resolution) to the secondary page.

B.2.6 color

color -changes screen color

black()
blue()
green()
red()
violet()

Any of the above routines will turn on the primary Hi-resolution graphics page in full screen mode, and also clear the screen.

black turns off all the dots.
blue turns off all the dots except the blue ones.
green turns off all the dots except the green ones.
red turns off all the dots except the red ones.
violet turns of all the dots except the violet ones.

B.2.7 plot

plot -plots points on the screen

plot (x, y)
int x, y;

bplot (x, y)
int x, y;

gplot (x, y)
int x, y;

rplot (x, y)
int x, y;

vplot (x, y)
int x, y;

The plot routines will plot a point at and given (x, y) location. The table below shows which routines plot which colored points.

plot plots a white point
bplot plots a blue point
gplot plots a green point
rplot plots a red point
vplot plots a violet point

The color plotting routines will plot the point if its location is within the limitations of the color on the hires screen.

B.2.8 mode

mode -selects display mode

text()
hgr()
fscreen()
mscreen()

text() sets the soft switch and returns you to text mode.
hgr() sets the soft switch and brings the screen to Hi-resolution mode.
hgr() does not clear the screen as the color routines will.

fscreen()

gives you a full screen to work with in the graphics mode. This mean the you have 280 by 192 matrix to work with.

mscreen()

sets the screen so that in the Hi-resolution mode you have a 4 line caption for normal text at the bottom of the screen. These 4 lines take up the bottom 32 rows of dots leaving a 280 by 160 matrix to work with.

Appendix C: DISKS

Manx Aztec C for Apple II DOS 3.3
SHELL version 2.4
from Manx Software Systems, 1983

Requires: 64k or larger Apple II with two 5.25" disk drives

There are nine diskettes:

Disk1_Start (not bootable as in the original package)
Disk1_StartBootable (boots DOS 3.3 and starts the SHELL)
Disk2_ARCH
Disk3_C65
Disk4_CCI
Disk5_LIB65
Disk6_LIBINT
Disk7_Graphics
Disk8_Save (mainly blank for saving programs)

These disks are distributed as AztecC_DOS33dsk.zip

Appendix D: Mini Manual Credits and Attributions

D.1 Rubywand

Thanks to Jeff Hurlbert (Rubywand) for the Manx Aztec C Mini-manual for DOS 3.3. That was quite a lot of scanning, ocr'ing and typing. The information in this manual came "from scanning/editing selected sections of the manual plus a few additions." According to Jeff.

Rubywand was active for years in the Apple II online community with such Apple II efforts as the The official Csa2 (comp.sys.apple2) Usenet newsgroup Apple II FAQs:

http://apple2.org.za/gswv/a2zine/faqs/A2FAQs5MAINHALL.html

His "contributorship" extends to other Mini Manuals besides this one:

http://apple2.org.za/gswv/a2zine/faqs/Csa2DOSMM.html

Rubywand's Aztec C Mini Manual is distributed in its original form as AztecC_minimanual.txt

The "published" date is Jan 13th 2001, with an update on April 28th, 2006. According to AztecC_minimanual.txt, the Aztec C65 Mini Manual was part of the GS WorldView Winter 2001 Issue.

I have no idea what happened to the original manual for this compiler. Rubywand's Mini Manual (this manual) is all we have left.

D.1.1 The Compiler

These disks are distributed as AztecC_DOS33dsk.zip

Along with Rubywand's distribution you may find different distributions of this same compiler on several different internet sites. Two of the sites are:

http://www.aztecmuseum.ca The Official Aztec C Online Museum

The purpose of this website is to provide a free internet archive for various versions of the now-discontinued Aztec C Compiler for older now-obsolete platforms, and to provide related compiler documentation and Aztec C source code and samples that support the **Fair Use** of these discontinued compilers for educational purposes by programmers, researchers and enthusiasts.

ftp://ftp.apple.asimov.net/pub/apple_II/ The Asimov Apple II Collection

The ASIMOV Apple II FTP Archive is the largest public repository of Apple II disk images and support materials related to the Apple family of home computers. Continually updated and loaded with thousands of disk images

The distribution of this compiler that RubyWand documents in this Mini Manual is the most complete distribution of this version in its original form that is known to still exist. The Aztec33 distribution extends Rubywand's to include a shared environment with an equivalent cross-compiler of the same vintage with extensions and extras.

Other distributions of the same version may not include the Graphics disk, and a single disk distribution (sides A and B) was even distributed as part of the White Disk Collection on disk 39:

ftp://ftp.apple.asimov.com/pub/apple_II/images/games/collections/White_Disks/

D.1.2 Other Attributions

When Rubywand updated this Mini Manual on April 28, 2006 he thanked Bill Malcolm for spotting an error.

Thanks to Michael J. Mahon for ocr error and other corrections around and before December 6, 2009.

I hope with this latest version I have got most of the few errors that remained sorted-out.

D.2 Manx Software Systems

Manx Software Systems of Shrewsbury, New Jersey, produced C compilers beginning in the 1980s targeted at professional developers for the Apple][(DOS 3.3 and ProDOS) and a variety of platforms up to and including PC's and Mac's.

Manx Software Systems was started by Harry Suckow, with partners Thomas Fenwick, and James Goodnow II, the two principle developers. They were all working together at another company at the time. Harry had started several companies of his own anticipating the impending growth of the PC market, with each company specializing in different kinds of software. A demand came for compilers first and he disengaged himself from the other companies to pursue Manx and Aztec C.

Harry took care of the business side, Fenwick specialized in front-end compiler development, and Goodnow specialized in back-end compiler development. Another developer, Chris Macey, worked with them for awhile on 80XX development and in other areas.

The name "Manx" was selected from a list of cats for no particular reason except that the name Harry wanted to use was taken by one of his other start-up companies.

One of the main reasons for Aztec C's early success was the floating point support for the Z80 compiler which was extended to the Apple II shortly after. Harry insisted on adding floating point.

During the move to ANSI C in 1989, Robert Sherry who was with Manx at the time and interested in the minutiae of standards represented them on the ANSI committee but left shortly after.

By this time Microsoft had targeted competitors for their C compiler and Aztec C was being pushed-out of the general IBM-PC compiler market, followed by competition with Apple's MPW C on the MacIntosh side and Lattice C on the Amiga after SAS bought them.

By the early 1990s Thomas Fenwick had left to work for Microsoft, and James Goodnow worked on Aztec C occasionally but was pursuing other projects outside the company

and eventually left the company altogether. Harry employed about 20 people at that time. Chris Macey returned as a consultant but eventually left to become chief scientist for another company. Mike Spille joined Manx as a developer along with the late Jeff Davis (embedded systems).

Throughout the 1990s they continued to make their Aztec C. As their market share dropped, they tried to make the move to specializing in embedded systems development, but it was too late. They disappeared a few years back following the loss of market presence of some of their target platforms (various 6502 machines, Atari and Amiga 68xxx, etc.).

In the end, Jeff Davis and Mike Spille helped Harry Suckow keep the company going before Harry finally closed it. Harry Suckow is still the Copyright holder for Aztec C.

Many professional developers used Aztec C compilers from Manx Software Systems before they vanished from the planet.

e.o.f. Rubywand 13Jan2001 amdg