Attribution

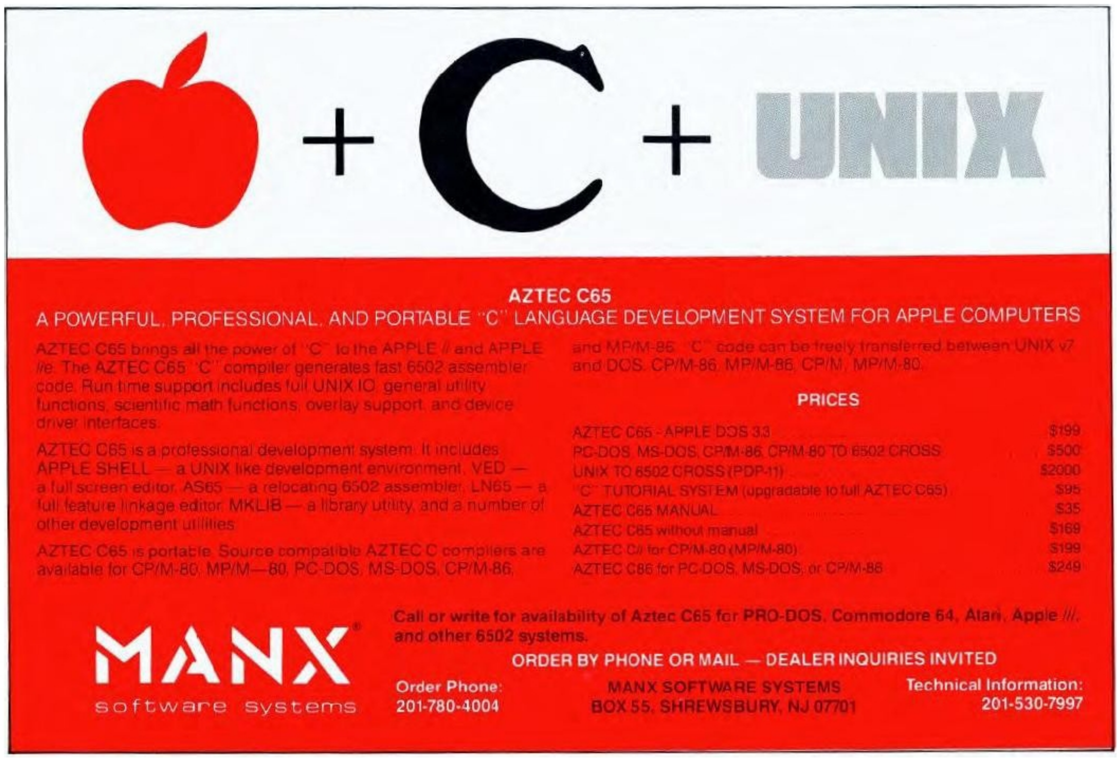
This document is excerpted from AztecC_minimanual.txt which was produced by Rubywand (Jeff Hurlburt). Despite the fact that I plagiarize Jeff's stuff shamelessly I can take no credit for his large body of work on behalf of the Apple II community.

In fact without Jeff's work in documenting the Apple II and the contribution of his cohorts from csa2 I could take little credit for my own work on the subject.

I have been formatting Rubywand's Aztec C Minimanual as part of a larger project. The Tutorial Introduction (this document) and Shell Sections are done so I am distributing those excerpts in advance of the whole manual.

All the Best,

Bill Buckels
bbuckels@escape.ca
August 2013



AZTEC C65
A POWERFUL, PROFESSIONAL, AND PORTABLE "C" LANGUAGE DEVELOPMENT SYSTEM FOR APPLE COMPUTERS

AZTEC C65 brings all the power of "C" to the APPLE // and APPLE //e. The AZTEC C65 "C" compiler generates fast 6502 assembler code. Run time support includes full UNIX IO, general utility functions, scientific math functions, overlay support, and device driver interfaces.

AZTEC C65 is a professional development system. It includes APPLE SHELL — a UNIX like development environment, VED — a full screen editor, AS65 — a relocating 6502 assembler, LN65 — a full feature linkage editor, MKLIB — a library utility, and a number of other development utilities.

AZTEC C65 is portable. Source compatible AZTEC C compilers are available for CP/M-80, MP/M—80, PC-DOS, MS-DOS, CP/M-86, and MP/M-86. "C" code can be freely transferred between UNIX v7 and DOS, CP/M-86, MP/M-86, CP/M, MP/M-80.

PRICES

AZTEC C65 - APPLE DOS 3.3	$199
PC-DOS, MS-DOS, CP/M-86, CP/M-80 TO 6502 CROSS	$500
UNIX TO 6502 CROSS (PDP-11)	$2000
"C" TUTORIAL SYSTEM (upgradable to full AZTEC C65)	$95
AZTEC C65 MANUAL	$35
AZTEC C65 without manual	$169
AZTEC C// for CP/M-80 (MP/M-80)	$199
AZTEC C86 for PC-DOS, MS-DOS, or CP/M-86	$249

Call or write for availability of Aztec C65 for PRO-DOS, Commodore 64, Atari, Apple ///, and other 6502 systems.

ORDER BY PHONE OR MAIL — DEALER INQUIRIES INVITED

MANX® software systems

Order Phone:
201-780-4004

MANX SOFTWARE SYSTEMS
BOX 55, SHREWSBURY, NJ 07701

Technical Information:
201-530-7997

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 Ile, 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 its 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-l);
  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.

<u>1.8 A Few Utilities</u>

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 still used the .REL 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 REL 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 Shell 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. Aztec 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.

<u>1.12 Going to the Source</u>

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 labelled 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.