Table of Contents

Forward

This document was produced by combining old documentation from Manx Software Systems about their Aztec C65 C compilers for the 8 bit Apple II platform, with my work and study of their C65 compilers that I have managed to collect over the years (and of course some “reverse engineering”). I have relied heavily on the work of others for some of this information and have made “best guess” assumptions through testing and observations where no information was available. Regardless of where it all came from, I believe all the information provided herein to be true, (and probably factual). The usefulness of all of this is quite another matter.

Aztec C65 - a Retro Compiler

The Aztec C65 compilers are genuine “retro-compilers” and part of a larger “family” of Aztec C compilers of their time, but are now part of history. They were commercial products and competed “in the real world”. They weren’t targeted at some modern crew trying to replace assembler and write smaller faster games or anything of that nature… they were targeted to the Application Developer of the day.

The C code doesn’t generate the fastest 6502 on the planet, but it’s not slow either, and the programs aren’t the smallest (in most cases), but aren’t huge, and are stable and feature rich. The language is old K&R C, which genuinely represents the era. The fact that Aztec C65 has persisted across the years and is every bit as usable today as it ever was attests to its excellence.

Having said that, today, some programmers prefer modern tools that all look about the same, and are not looking for a retro experience at all.

Aztec C65 is not an optimizing compiler. Responding to market demand as Manx Software reached the end, some versions of Aztec C were shipped with a third party optimizer; called “COP” by Research in Motion (the same company that developed the “BlackBerry”). But Aztec C65 (like the Apple II) didn’t survive in the mainstream long enough to bother with COP… there was no money in it anymore and that’s how things work in the “real world”.

In the competitive “real world” compiler market of the day, the source code of a compiler, assembler, and linker was considered proprietary. Consequently the programmers of the day didn’t tweak compilers… they couldn’t so they tweaked code. Aztec C65’s “core” source code was never released, (and today it is lost). So Aztec C is now like Latin, for me and others like me who like to use old compilers to write programs for old computers.

However, Manx Software Systems released the source code for just about everything else that came with these compilers (runtime libraries and utilities), and created extensive technical documentation throughout the 1980’s and into the 1990’s when they vanished from the planet, along with Aztec C. So we essentially have the source for everything else, and extensive knowledge about using Aztec C and its internal workings and many features, but not the ability to change their compilers.

Aztec C65 uses a scheme called a "PSEUDO STACK" to push and pop the stack (variables etc) for each function call and return. This creates larger (and slower) programs than pure machine code programs and the like. But if it's machine code you want, Aztec C65 supports inline assembly and open inline subroutine calls, and linking to 6502 assembly modules.

Aztec C65 Compilers Currently Available

Only 2 "vintages" of Aztec C65 compilers are currently available from the Aztec C65 Website:

- CII C65 DOS 3.3 Compiler version 1
- C65 ProDOS and DOS 3.3 Compiler version 3.2

The different "bundles" (variations) of these 2 compilers are based on these 2 "vintages".

Overview of Aztec C65 Compilers

Aztec C65 compilers for the Apple II come in 2 "flavours":

- MS-DOS Cross-Compilers (Run in Windows or an emulator like DOSBox)
- Apple II Native-Mode (Run on a Real Apple II or in Emulators like AppleWin)

Aztec C65 compilers produce executable programs for 2 targets:

- "RAW" (SA –Stand-Alone) executables (DOS 3.3 and ProDOS)
- PCODE (Pseudo-Code) executables that run only in the Apple II Aztec C Shells

But note also that PCODE executables can also contain "RAW" functions to speed-up time critical portions. These "hybrid" programs will not run outside Aztec C's Unix-like Shells, but are smaller than "RAW" Apple II programs, and support the features of the Shells, including command-line arguments.

Aztec C65's Unix-like Shells

Aztec C's Unix-like Shell for the Apple II came in 2 different releases. These two shells are quite different; the earlier release is a DOS 3.3 "RAW" BRUNable BIN program called "SHELL". The later release is a ProDOS 8 SYS program called "SHELL.SYSTEM":

DOS 3.3 APPLE][SHELL 2.4 1984

SHELL 2.4 was released as part of the CII compiler for DOS 3.3. Although the Aztec C65 documentation suggests that the newer ProDOS compiler C65 3.2b can produce PCODE executables that run in the DOS 3.3 shell, this is not really correct. PCODE executables produced with the newer compiler are not 100% compatible with the older shell; screen controls do not work properly, command-line args don't work, and the newer compiler's PCODE executables are much larger than 100% compatible PCODE executables produced by the older CII compiler. Only the older CII DOS 3.3 compiler produces programs that are truly compatible with the DOS 3.3 SHELL version 2.4.

More about the DOS 3.3 Shell (and the CII compiler) can be found at the following link:

http://www.aztecmuseum.ca/docs/AztecC_minimanual2013.pdf

ProDOS 8 Aztec C Shell v1.99j 6-27-86

The Aztec C Shell for ProDOS is more advanced and has more features than the older DOS 3.3 Shell, but the PCODE executables are larger.

The Manual is at the following link:

http://www.aztecmuseum.ca/docs/SHELLC65A.pdf

Version 1 – the CII compiler for DOS 3.3

http://bitsavers.informatik.uni-stuttgart.de/pdf/manx/Aztec_C_Data_Sheet_Mar83.pdf

Aztec C65 for the Apple II was first released for Apple Computer's DOS 3.3 in 1982 even before the IBM-PC's Aztec C86. The ProDOS operating system did not exist until the following year. The Aztec C65 Commodore 64 compiler was also based on this compiler.

More about the CII compiler for DOS 3.3 can be found at the following link:

http://www.aztecmuseum.ca/docs/AztecC_minimanual2013.pdf

Version 3.2 – the C65 ProDOS and DOS 3.3 Compiler

This was the last Aztec C compiler released for the Apple II. Version 3.2b was released as a cross-compiler and native mode compiler both, but the native mode version did not support floating point for "RAW" DOS 3.3 and ProDOS SYS programs because it was not properly compiled. Like any "real-world" program, bugs can still exist in an old compiler. For example, I recently patched a patch provided by Manx to the 3.2b cross-compiler to allow a ProDOS SYS program to exit cleanly.

The CG65 version 3.2c cross-compiler which also supports ROM based code is also based on 3.2b, and supports other 6502 platforms like the Commodore 64 and Atari 400/800 but the libraries for the other platforms are not available…

So at this point in time, the C65 3.2b cross-compiler is the most complete, pending my further research etc.

Aztec C65 version 3.2b (and Aztec CG65 version 3.2c) will produce the following executables:

- ProDOS SYS programs
- ProDOS and DOS 3.3 BRUNable BIN programs
- ProDOS PCODE Shell programs

These include OVERLAY programs.

The Aztec C65 Assembler (AS65) will also produce pure BRUNable assembly language programs for the Apple II (if you know how to use it). These can also be loaded and called from within an Aztec C65 program.

As mentioned earlier, despite what the Aztec C documentation says, the DOS 3.3 Shell does not behave nicely when running PCODE programs written in 3.2b so use the older CII compiler to make those instead.

Documentation for the 3.2b cross-compiler is not currently available, but the manual for the native mode 3.2b compiler is and is almost the same. The CG65 3.2c cross-compiler manual is also almost the same.

The Aztec C Website has documentation for both of these, and also an html manual of sorts:

http://www.aztecmuseum.ca/docs/index.htm

About this Document – Aztec C65 3.2b and CG65 3.2c Utilities

My motivation for creating this document and the associated project that it is bundled with is simply a desire to provide some info for those brave souls who wish to get a little more in-depth into peripheral interfacing in an Aztec C65 Apple II program.

This document describes 3 utilities that, like Aztec C65, were under development until Manx Software Systems vanished from the planet. These utilities are provided with source code. You could read the source code (and run these utilities) and attempt to unravel their hieroglyphic mysteries.

But this document provides a saner (and more useful, productive, and less-cryptic) alternative to the fate that would surely ensue by "wading-into" a virtual sea of retro minutiae without the required spells and chants that are described herein.

Manx's "official" description of these 3 utilities (from photocopies they "shoved into" the back of the CG65 3.2c manual):

- CONFIG – Device Configuration Program
- TTY – Terminal emulator that runs on the Apple II
- XFER – File Transfer Program

Apple II ProDOS 8 Shell versions of all 3 of these utilities were provided (on disk 4) with the C65 3.2b compiler. Apple II DOS 3.3 "RAW" versions of TTY and XFER were also provided on a separate disk (disk 5).

One of these utilities (CONFIG) is a hardware set-up program for Aztec C65 programs. CONFIG.EXE can be run in MS-DOS/Windows. CONFIG works for "RAW" ProDOS 8 SYS and BIN programs and "RAW" DOS 3.3 BIN programs produced with the C65 3.2b compiler (and also with "RAW" programs produced by CG65 3.2c.

The other two utilities (TTY and XFER) deal with the COM port on the Apple II.

The manual for C65 version 3.2b's native-mode compiler on the Aztec C Website includes documentation for CONFIG and TTY (but not for XFER which was never distributed with the native mode compiler):

http://www.aztecmuseum.ca/docs/MANXC65A.PDF

In the native-mode version of Aztec C65 3.2b there was no need to transfer programs from MS-DOS on a PC to a DOS 3.3 disk (using a NULL modem cable). Manx said to use the ProDOS CONVERT program provided by Apple Computer. Since the native-mode compiler ran in ProDOS there also was no need to transfer files to ProDOS (of course).

For the CG65 compiler, Manx Software Systems distributed the documentation in text format in their “source archives”. The source code and binaries are identical to those distributed (without text file documentation) with the C65 version 3.2b cross-compiler. It is this text file documentation that provides the content for most of the rest of this document.

In today’s world, it might seem odd that so much effort has been put into these types of low-level utilities. It was a different computing world back then (throughout the 1980’s) of course.

The XFER program in particular was carried across the platforms that Aztec C supported, including the Commodore 64. XFER still stands as an example of how it could be done back then, or even now, if you wanted to.

But by the late 1980’s, we developers used Communications Programs on both sides of our MS-DOS PC’s and our Apple II’s with our null-modem cables, and protocols like YMODEM, to transfer batches of files (instead of one file at a time).

These (third-party) Communications Programs that we used also provided terminal support, so TTY is obviously somewhat redundant as well, unless from a learning perspective (or a retro-developing perspective).

As far as configuring Aztec C65 Apple II programs to make use of peripherals and slot devices; if you have such things, potentially CONFIG may be useful.

But for whatever use any of this is, the rest of this document presents the original and last known information for these 3-utilities from Manx Software Systems.

I have also compiled these 3 utilities as faithfully as I could, cleaning-up the makefiles and so forth. They never did compile “out of the box”. Manx compiled them internally and distributed them in binary format with their unvarnished source. The programmer who received them was expected to make any adjustments necessary to re-compile.

Bill Buckels
bbuckels@mts.net
November 9, 2013

Dedication:

Dedicated to Riccardo Greco for learning to develop in Aztec C65 for the Apple II in this day and age.

NAME - TTY

tty - terminal emulation program

SYNOPSIS

```
tty -syy -bxx
```

DESCRIPTION

tty is a terminal emulation program that allows an Apple // operator to talk to another computer. To the other system, the Apple // will appear to be a terminal that supports some of the special features of the ADM-3A terminal.

tty reads characters from the keyboard and writes them to a serial interface. It also reads characters from this interface and writes them to the console. This interface must be compatible with the Super Serial Card.

The -s option defines the number of the slot containing the interface. The number immediately follows the -s, with no intervening spaces. If this option isn't specified, the interface is assumed to be in slot 2.

The -b option defines the baud rate of the serial interface. The baud rate immediately follows the -b option, with no intervening spaces. If this option isn't specified, the baud rate is assumed to be 9600.

To exit tty, type control-2; ie, type the '2' key while holding down the control key.

NAME - CONFIG

config - Define device attributes

SYNOPSIS

config [file]

DESCRIPTION

config defines device attributes to a program that was created using Aztec C. It does this by modifying, as directed by you, the table that defines these attributes.

This description of config first describes the device table, then how to use config, and finally gives some examples of config usage.

1. The Device Table

The device table used by the SHELL and by programs of type PRG (ie, programs that can only be run in the SHELL environment, because they have been linked with the shmain module instead of the samain module), resides in the SHELL's memory-resident environment area. This area, which contains information that the SHELL needs to maintain between program executions, is loaded from the file that contains the SHELL. It is loaded only when necessary: when the SHELL is first loaded and when the SHELL detects that its environment area has been corrupted. PRG programs and BIN programs created using Aztec C won't corrupt the SHELL's environment area unless they go out of their way to do so, and the SHELL tries to keep programs that were created without using Aztec C from corrupting this area by setting the HIMEM field to the base of this area. So when you're doing standard development activities under the SHELL, the SHELL's memory-resident environment area (and hence the SHELL's memory-resident device table) is normally only loaded once: when the SHELL is first loaded.

A program created using Aztec C that either runs under ProDOS and is of type BIN and SYS (ie, that can run outside the SHELL environment) or that runs under DOS 3.3 uses a device table that resides in the program's memory space. This table is loaded along with the program from a copy in the file from which the program is loaded.

config can modify the copy of a program's device table that is in the file that contains the program. For example, the SHELL is just a program created using Aztec C, and you can modify the device table that is within the file that contains the SHELL, shell.system. The modifications that you make to this copy of the SHELL's device table will take affect (ie, affect executing SHELL and PRG-type programs) when the SHELL next reloads this table into its memory-resident environment area.

config can also directly modify the SHELL's memory-resident device table, without modifying a file-resident device table. This feature allows you to temporarily redefine device attributes; you can modify a program's file-resident table when you want to make a more permanent redefinition of device attributes.

Before getting into the description on config usage, we are going to define the names by which you will refer to devices. Then, since config lets you access a device table at a low level, allowing you to examine and modify specific fields and bits, we define the device table in detail.

1.1 Device Names

The following list defines the names of devices:

- con: The console device
- pr: The printer
- ser: The serial device
- sx: The device that's in slot x. For example s3: is the name of the device in slot 3.

1.2 Device Table Organization

The include file device.h contains definitions related to devices. This section defines information that is in that file.

A device table has the following structure:

```
struct _dev_info {
 char fnd_str[14];     /* signature */
 short con_flags;      /* console flags */
 struct sgttyb tty;    /* con info for ioctl */
 struct _name_dev
 dev_con,              /* con: info */
 dev_pr,               /* pr: info */
 dev_ser;              /* ser: info */
 struct _slot_dev
 slots[8];             /* slot info */
 int init_max;         /* init space size */
 int init_len;         /* init space used */
 char init_buf[];      /* init space */
};
```

1.2.1 The fnd_str Field

The fnd_str field contains a (hopefully) unique character string; when told to find a file's device table, config looks for this string.

1.2.2 The con_flags Field

The con_flags field contains bits that define console attributes. the following tables lists for each bit its symbolic name, a value in parentheses that defines its location in the field, and the meaning of the bit.

CON_IMAP (0x01)

When this bit is set, the console driver will perform keyboard mapping. This mapping is needed needed for Apples whose keyboards don't support the full ASCII character set.

CON_UPPR (0x02)

When this bit is set, the console driver will turn lower case characters into upper on output.

CON_HIGH (0x80)

When this bit is set, the console driver will set the high order bit of each character it outputs.

The config commands dump con: and mod con: are used to display and modify the con_flags field.

1.2.3 The tty Field

The tty field contains those fields related to a program's console I/O that the program modifies by calling the ioctl function. For example, this field defines whether the console is in line-oriented or character- oriented mode. config doesn't modify this field, so we won't discuss it further here. For a complete discussion of console I/O and ioctl, see the Console I/O section of the Library Overview chapter.

1.2.4 The dev_con, dev_pr, and dev_ser Fields

The dev_con, dev_pr, and dev_ser fields define the slot devices that are associated with the con:, pr:, and ser: devices, respectively. These three devices are just generic names that allow a program to access the console, printer or serial card without having to know the exact slot that contains the card. When a program performs I/O to one of these generic devices, the I/O operation is simply passed on to its associated slot device. For example, if a program writes to pr:, and during one execution the device table used by the program specifies that pr: is associated with slot device s1:, then the data will be written to the card in slot 1. If during another execution the table specifies that pr: is associated with slot device s2:, then the data will be written to the card in slot 2.

The config commands dump con:, dump pr:, and dump ser: display the slot device that's associated with a generic device. And the commands mod con:, mod pr:, and mod ser: define the slot device that's associated with a generic device.

The console device driver does not currently look at the dev_con field; it always performs console input by calling the RDKEY ROM routine that begins at 0xfd0c, which in turn calls the console input routine whose address is in the KSW zero-page field. It performs console output by calling the COUT ROM routine that begins at 0xfded, which in turn calls the console output routine whose address is in the CSW zero-page field. If the Apple on which the SHELL is running has an 80 column card, the SHELL will automatically set up the CSW and KSW zero page fields to use it.

1.2.5 The slot Fields

The slot field is an array of eight structures of type struct _slot_dev, each of which defines the attributes of one slot device. This structure will be defined after we finish describing the fields in the device table structure.

1.2.6 The init_buf, init_max, and init_len Fields

When a program opens a slot device, the slot device driver can optionally send an initialization string to the device. The following _dev_info fields define information related to device initialization strings:

- init_buf contains the strings;
- init_max defines the length of the init_buf space;
- init_len defines the number of bytes in init_buf that have been allocated to initialization strings.

config allocates an initialization string by placing the string at the current location defined by init_len and then incrementing init_len past the string.

The size of the SHELL's init_buf area is hard-coded into the SHELL to be 64 bytes; this limit can't be exceeded. The size of a BIN, SYS, or DOS 3.3 program's init_buf area is also hard-coded to be 64 bytes, but you can change this by modifying and replacing the devtab module in the various versions of c.lib (c.lib, ci.lib, d.lib, and di.lib).

The following config commands access the string initialization
fields:

- The dump init command displays the contents of the init_max and init_len fields.
- The mod init command resets the init_len field and turns off each slot device's INIT_STR bit, thus freeing all of the string initialization space.
- The commands that display and modify information about slot devices (dump sx: and mod sx:) access the device table's string initialization fields.

1.3 The _slot_dev Structure

A slot device's struct _slot_dev structure has the following form:

```
struct _slot_dev {
 short outvec;         /* CSW vector ($36-37) */
 short invec;          /* KSW vector ($38-39) */
 short  init;          /* init str offset in initbuf */
 char slot;            /* $s0 */
 char hi_slot;         /* slot number */
 char type;            /* -1=BASIC, 0=Pascal1.0, 1=Pascal1.1 */
 char flags;           /* slot attr flags*/
 char tabp;            /* line pos for tab mapping */
 char tabw;            /* tab width */
 char iflags;          /* slot init flags */
 char xtra;            /* unused */
};
```

1.3.1 The outvec and invec Fields

The outvec and invec fields in a slot device's _slot_dev structure contains the addresses of the routines that the slot device driver will call to transfer each character to or from the slot device, respectively. Normally, the slot device driver sets these addresses to locations within the associated card's ROM space at the time the device is opened, by determining the type of protocol (Basic, Pascal 1.0, or Pascal 1.1) used by the ROM routines. As discussed below, the setting of these fields for a particular slot device by the slot device driver can be disabled by turning off its INIT_VEC bit in the iflags field of its _slot_dev structure.

config doesn't have commands to modify these fields, although the dump sx: commands display their contents.

1.3.2 The init Field

The init field in a slot device's _slot_dev structure contains the offset within the _dev_info structure's init_buf area at which the device's initialization string begins.

As discussed below, when a slot device is opened, the INIT_STR bit in the iflags field of the device's _slot_dev structure determines whether the device's initialization string should be sent to it.

The dump sx: command will display the initialization string that's associated with slot device sx:, and the mod sx: str=... command will assign an initialization string to sx:.

1.3.3 The type Field

The type field in a slot device's _slot_dev structure defines the type of protocol used by the device's ROM routines. It can have the following values:

Value	Meaning
-1	Basic
0	Pascal 1.0
1	Pascal 1.1

This field is normally set when the device is opened, but if the INIT_VEC bit in the iflags field of the device's structure is off, the type field is not set.

The dump sx: command will display sx:'s type field, but config doesn't have a command to set it.

1.3.4 The slot and hi_slot fields

The slot field in a slot device's _slot_dev structure defines the high-order byte of the address of the device's ROM space. The hi_slot field defines the number of the slot.

The dump sx: command will display sx:'s slot and hi_slot fields, but config doesn't have commands that will modify these fields.

1.3.5 The flags Field

The flags field in a slot device's _slot_dev structure contains bits defining various attributes of the slot. The following table lists for each bit the symbolic name of the bit, a value in parentheses that defines the location of the bit, and the bit's meaning.

SLOT_LFCR (0x01)

When this bit is set, an carriage return character that is read from the device will be translated to a linefeed, and a linefeed character that is written to the device will be translated to a carriage return.

SLOT_TABS (0x02)

When this bit is set, an output tab character will be replaced by enough space characters to position the device at the next "tab stop". For more information on tab stops, see the descriptions of the tabw and tabp fields.

SLOT_UPPR (0x04)

When this bit is set, an output alphabetic character will be converted to upper case.

SLOT_CRLF (0x08)

When this bit is set, and a program writes a carriage return character to the device, the device driver will also write a line feed character to the device.

SLOT_HIGH (0x80)

When this bit is set, the high order bit will be set on all output characters.

The dump sx: command will display in binary the contents of sx:'s flags field, and the mod sx: command can be used to modify it.

1.3.6 The iflags Field

The iflags field in a slot device's _slot_dev structure contains bits that define attributes of the device. The following table lists for each bit the symbolic name of the bit, a value in parentheses that defines the location of the bit, and the bit's meaning.

INIT_VEC (0x01)

When this bit is set and the slot device is opened, the addresses of its input and output vectors will be determined and set in the structure's invec and outvec fields based on the type of protocol used by the card, and the type field will be set.

INIT_CAL (0x02)

When this bit is set and the slot device is opened, the card will be initialialized by issuing a call to the first byte of its ROM space.

INIT_STR (0x04)

When this bit is set and the slot device is opened, the device's initialization string (which is pointed at by the init field of the device's _slot_dev structure) will be written to the device.

INIT_ONCE (0x08)

When this bit is set, the initialization activities described above will only be performed once for the slot device.

The dump sx: command will display in binary the contents of sx:'s iflags field, and the mod sx: command can be used to modify it.

1.3.7 The tabw Field

The tabw field in a slot device's _slot_dev structure defines the number of characters between each of the device's tab stops. If 0, it's assumed to be eight.

When a tab character is written to a slot device, the slot device driver can optionally output spaces in its place until the next tab stop is reached. This replacement is enabled by setting the SLOT_TABS bit in the flags fields of the device's _slot_dev structure.

The dump sx: and mod sx: commands can be used to display and modify the contents of sx:'s tabw and tabp fields.

1.3.8 The tabp Field

The tabp field in a slot device's _slot_dev structure defines the number of characters that have been send to it since the last carriage return or linefeed. When a tab character is sent to a device, the device driver uses the device's tabp field to decide how many spaces it should output to reach the next tab stop.

1.3.9 The xtra Field

This field is unused.

2. Using config

config is an interactive program: you enter commands to it to examine device information, make modifications, and so on. These commands access a copy of the table that resides in an internal buffer within config: when you define the file whose device table you want to modify (or the table that's in the SHELL's memory-resident environment area), config finds the table and reads it into its internal buffer; your examination and modification commands then access the table that is in this internal buffer. When you've completed the modifications, you type the write command, which causes config to write the table back to the location from which it was obtained.

The location of a table to be modified can be specified to config as an argument when it is started. This argument can be the name of a file containing a program whose device table is to be modified. It can also be the word mem:, which specified that the SHELL's memory- resident device table is to be modified. When config is started with this optional argument, it automatically searches for the table in the specified location and, if found, reads it into its table; if config doesn't find the table, it will tell you.

Alternatively, the location of a device information table that's to be modified can be specified once config is active, by typing the open command. This command takes a single argument: the name of the file whose device table is to be modified; or mem:, if the SHELL's memory-resident table is to be modified. The open command searches for the table in the specified location, but doesn't load it into config's internal buffer; if you want the table read, you must explicitly say so, using config's read command. This allows you to make changes to one program's device table and then write the modified table to several different programs.

3. Commands

config has just a few basic commands, most of which have arguments. We'll first list the commands and give a brief description. The following paragraphs will then discuss the commands in detail.

Command	Description
dump	Display information
mod	Make modifications
open	Prepare to examine/modify a device table
read	Read a device table into config's internal buffer
write	Write a device table from config's internal buffer
quit	Halt config

3.1 The dump Command

The dump command displays information about one device, about all devices, or about the device initialization string space. The command has the following format:

dump [dev]

The optional argument dev is the name of the device about which you want to get information, or the word init if you want information about string space. If an argument isn't specified, information about all devices and about string space is displayed.

3.2 The mod Command

The mod command is used to modify device attributes and to reset the device initialization string space. The command has the following format:

mod dev [args]

where dev is the name of the device whose attributes are to be changed, or init if string space is to be initialized; [args] are arguments defining the attributes that are to be changed. The arguments to the mod command depend on the device being modified. The following paragraphs discuss the modification of each device.

3.2.1 Modifying con:

The command for modifying con: has the following format:

mod con: [flags=xx] [imap] [uppr] [high]

where square brackets surround optional arguments. imap, uppr, and high usually cause a bit to be turned on in the con_flags field. If preceded by a ~ character, they cause the designated bit to instead be turned off.

The arguments have the following meanings.

Argument	**Meaning**
flags=xx	sets the field in the device table that defines console attributes, con_flags, to the hex value xx.
imap	set (or reset, if preceded by ~) the CON_IMAP bit in con_flags.
uppr	set or reset the CON_UPPR bit in con_flags.
high	set or reset the CON_HIGH bit in con_flags.

3.2.2 Modifying pr: and ser:

The commands that modify the pr: and ser: attributes are similar:

mod pr: sx:
mod ser: sx:

where sx: is the name of the slot device that is to be associated with pr: or ser:.

3.2.3 Modifying Slot Devices

The command for modifying the attributes of a slot device has the following form:

mod sx: [flags=xx] [iflags=xx] [tabw=dd]
[lfcr] [crlf] [tabs] [uppr] [high] [cal] [once] [vec]
[str=string]

where sx: is the name of the slot device.

3.2.3.1 The flags and iflags Arguments

The flags=xx argument sets the device's flags field to the hex value xx.

Similarly, the iflags=xx argument set the device's iflags field to the hex value xx.

3.2.3.2 The lfcr, ... Arguments

The arguments specified on the second line (lfcr, ...) usually cause a bit in the device's flags or iflags field to be turned on; if preceded by a ~ character, they instead cause the designated bit to be reset. The symbolic name of the bit represented by these arguments can be derived by appending "SLOT_" (for a flags bit) or "INIT_" (for an iflags bit). For example, the command "mod s2: lfcr" sets the bit SLOT_LFCR in the flags field in the _slot_dev structure for the s2: device.

3.2.3.3 The str=string Argument

The argument str=string usually sets the initialization string for the specified slot device to string and turns on the INIT_STR bit for the device. If the argument has the form ~str (ie, preceded by a ~ character and not followed by =string), the INIT_STR bit is instead turned off.

Strings are usually specified by surrounding them with double- quote characters, although if a string contains just printable characters with no spaces it can be specified without the surrounding quotes.

In a quoted string, a printable character is represented by itself. Unprintable characters are represented by a sequence that begins with a backslash character, as defined in the following table.

Sequence	Meaning
\n	Newline
\t	Horizontal tab
\b	Backspace
\r	Carriage return
\f	Form Feed
\\	Backslash
\xyy	The hex value yy

For example, the command to set the initialization string for slot device s2: to a string consisting of an escape character (hex value 1b), followed by the character Q would be:

mod s2: str="\x1bQ"

3.2.4 Resetting Device Initialization String Space

The following command resets the use of the space used for device initialization strings:

mod init

It resets the field that points to the top of allocated string space (init_len in the device information structure), and turns off the INIT_STR bit and clears the init field for each slot device.

3.3 The open Command

The open command prepares config for accessing a device table that is in a file or in the SHELL's memory-resident environment area. The command has the following form:

```
open file
```

where file is the name of the file whose device table is to be accessed, or mem: to access the table in the SHELL's memory-resident environment area.

If a file is specified, the open command causes config to open the file and search for the file's device information table. If the table is not found, config will say so.

Note that the open command does not read the specified device information table into config's internal buffer; you must explicitly tell config to do that, using the read command.

3.4 The read Command

The read command causes config to read the device information table from the currently open file or SHELL environment area into config's internal buffer.

Note that when a file or the SHELL's environment area is specified as a command-line argument, config automatically reads the device information table into its internal buffer, making it unnecessary to issue a read command.

3.5 The write Command

The write command causes config to write the device information table that's in its internal buffer to the currently open file or SHELL environment area.

3.6 The quit Command

The quit command causes config to halt.

4. Examples

In this example, the device table in shell.system is modified for use with an Image Writer printer that's connected to a Super Serial card in slot 2. The changes are also written to the SHELL's memory-resident table. First, we get config started by entering the following command to the SHELL:

```
config shell.system
```

Once started, config finds the device table that's in the file shell.system and reads it into config's internal buffer.

We next display the current settings of s2:'s flags and fields by
entering to config:

dump s2:

We next set and reset flags and fields that define how i/o to s2: is to be performed. These changes are made just to the device table that's in config's internal buffer; they won't be made to the device table that's in shell.system until the write command is issued.

mod s2: vec cal ~lfcr tabs tabw=4 str="\x1bQ"

The operands to the mod command have the following meanings:

- vec tells the device driver to determine the addresses of s2:'s ROM routines when s2: is opened;
- cal tells the driver to call the device's initialization code, which begins at the first byte of its ROM, when the device is opened;
- ~lfcr tells the driver not to send a linefeed after a carriage return;
- tabs tells the driver to output spaces in place of tabs;
- tabw=4 says that there are 4 spaces between tab stops;
- str="\x1bQ" tells the driver to send the specified character string to the device when it is opened.

We next display the modified settings of s2:'s flags and fields, which are in the device table that's in config's internal buffer:

dump s2:

Everything's OK, so we write the modified device table back to shell.system:

write

The changes that have just been made won't affect an executing SHELL or PRG programs until the SHELL detects that its environment area has been corrupted, or until the SHELL is loaded following system power-up. To make these changes take effect immediately, we need to write them to the SHELL's memory-resident device table. To do this, we first tell config that we want to access this memory-resident table by entering:

open mem:

This doesn't affect the device table that's in config's internal buffer, so we can immediately issue the following command to overwrite the SHELL's memory-resident table:

write

We're all done, so we exit config by entering:

Quit

Config History

Aztec C65 CII Version 1.05i 6502 © Copyright 1982, 1983 by Manx Software Systems provided an earlier version of the CONFIG utility that came with the C65 3.2 release. This earlier version did not configure all Aztec C65 Standalone programs; it was designed only to configure the DOS 3.3 SHELL to use peripherals.

By the time Aztec C65 3.2 was released, the idea behind the original config program had been expanded to include configuring all Aztec C65 programs including the later ProDOS Aztec C65 SHELL.SYSTEM (the "new" SHELL) and included more configuration options.

What follows is the Aztec C65 documentation for the earlier CONFIG program from which the later version "descended". The later version of CONFIG includes setting of tab widths. In the earlier version, a separate TABSET program was used to set the tab width for the SHELL. Documentation for the TABSET program follows ahead of the earlier CONFIG documentation.

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.

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.

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.

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.

NAME - XFER

xfer - file transfer utility

SYNOPSIS

xfer [-r] [+abpr] srcfile [destfile]

DESCRIPTION

xfer transfers a file from one machine to another over a serial link. To do so, xfer must be run on each of the two machines and a cable connected between the two serial ports. One machine is considered to be the master and the other the slave, with the master telling the slave what to do. Whether a machine is the master or slave is defined by specifying or not specifying arguments, respectively, when its xfer is started.

srcfile is the name of the file that is to be sent, and destfile is the name of the file that is to be received. if destfile isn't specified, it's assumed to be the same as srcfile.

1. Options

The arguments to xfer are as follows:

+a Specifies that the file being transferred is an ASCII text file. When this option is used, line endings are converted as necessary. For example, if the sending machine uses a carriage return-line feed combination to terminate a line of text, while the receiving machine uses a newline character for this, this translation will be made when the file is transferred.

+b When a file is being transferred to ProDOS, +b specifies that on ProDOS the file type should be set to BIN. For more information, see below.

+r When a file is being transferred to or from the Macintosh, +r specifies that on the Macintosh the file's resource fork is to be accessed.

+p Specifies that a program is being transferred.

-r Specifies that the file is to go from the slave to the master. If this option isn't used, the file is sent from the master to the slave.

When the +a, +b, and +p options are used, the file is assumed to contain arbitrary binary data.

2. Machine-dependent Information

2.1 Information for Macintosh

- Data is sent through Macintosh port A.
- ASCII text is sent to or from a Macintosh using the +a option. This option causes line endings to be converted as described above. When data is sent to the Macintosh, the +a option also causes the Macintosh file's type and creator to be set to TEXT and Manx.
- A program is sent to or read from a Macintosh using the +p option. When a program is received by the Macintosh, the +p option causes the type and creator fields of the Macintosh file to which the program is written to be set to AZTC and Manx. When a program is sent from the Macintosh, the +p option has no effect on the Macintosh (but it might, of course, have an effect on the other computer).
- The +r option causes the resource fork of the specified Macintosh file to be accessed. If this option isn't specified, the Macintosh file's data fork will be accessed.
- When data is sent to or read from a Macintosh without using the +a +p options, the Macintosh file's type and creator are set to TEXT and ????.
- For example, to send ASCII text to the data fork of a Macintosh file, you would use the option +a. To send a program to the resource fork of a Macintosh file, you would use the options +pr.

2.2 Information for Apple //

There are two versions of xfer for use on an Apple //, one which runs under ProDOS in the SHELL environment, and one which runs under DOS 3.3. In both cases, the Apple // serial card must be in slot 2 and must be compatible with the Super Serial card.

The following paragraphs first discuss the special features of the ProDOS version and then of the DOS 3.3 version.

2.2.1 ProDOS Information

- The +a option is used to send ASCII text. When data sent using the +a option is received by an Apple //, line endings are converted as decribed above, the type of the file to which the received data is written is set to TXT, and the file's AUX TYPE field is set to 0.

- To send to ProDOS a program that can only be executed in the SHELL environment, the +p option is used. In this case, the data, less the first four bytes (which contain control information), is written to a file of type PRG. The program's load address, which is in the data's first two bytes, is set in the file's AUX TYPE field. (Note: if the program's load address is 0x2000, the file's type is set to SYS instead of PRG).

- To send to ProDOS a program that can be executed either inside or outside the SHELL environment, the +b option is used. In this case, the data, less its first four bytes (which contain control information), is written to a file of type BIN. The program's load address, which is in the data's first two bytes, is set in the file's AUX TYPE field. (Note: if the program's load address is 0x2000, the file's type is set to SYS instead of BIN).

- To send arbitrary binary data (ie, data other than ASCII text and programs) to ProDOS, don't specify the +a, +p, or +b options. In this case, the type of the file to which the data is written is set to 0.

2.2.2 DOS 3.3 Information

- The DOS 3.3 version of xfer can only be used in 'slave' mode.

- When a file is sent to Apple // DOS 3.3, the type of the file is set to TXT if the +a option was specified; otherwise, the type is set to BIN.

2.3 Information for Commodore 64

- When data is sent to a Commodore 64 using the +p option, the second two bytes of data are not placed in the Commodore file.

2.4 Information for Amiga

- Data is sent through the serial port on the Amiga.

3. Examples

For all of the following examples, the slave machine's xfer is started by typing

xfer

In fact, the absence of arguments when starting xfer defines the machine on which this xfer is started as being the slave machine.

The following command, when executed on the master machine transfers the file datafile from the master machine to the slave machine. The file is assumed to contain arbitrary binary data, and no translations or manipulations are done to it.

```
xfer datafile
```

The next command, when executed on the master machine, transfers the source file test.c from the master machine to the slave machine. The name of the file that's created on the slave machine is also test.c.

```
xfer +a test.c
```

The next command, when executed on the master machine, transfers a program from the master to the slave. The name of the file on the master is test. On the slave, the program is placed in the /bin directory in a file called tprog.

```
xfer +p test /bin/tprog
```

The next command, when executed on the master machine, transfers text from the file named test.out on the slave to the file named output on the master.

```
xfer -r +a test.out output
```

4. Baud rate

The default baud rate is 9600, but can be changed by recompiling with the macro BAUD defined to the appropriate value.

5. Source

The source to the XFER program is provided with the package for adaptation to different hardware configurations. Included in the source are drivers for the Macintosh, Apple // Super Serial Card, Amiga, IBM PC Asynchronous Communications Adapter, and the Interfacer 4 from Godbout.

xfer consists of a system-independent module, whose source is in xfer.c, and a machine-dependent module. To generate a version of xfer for a particular machine, compile xfer.c and the machine- dependent module and link the two together.

6. Problems

When xfer is used between two machines whose speeds greatly differ, you may need to put a timing delay in the faster machine's xfer. To do this, rebuild the faster machine's xfer; when you compile its xfer.c, specify a value for the DELAY macro using the compiler's -D option. Begin with a value of 50 for DELAY, and work up from there. For example,

```
cc -DDELAY=50 xfer.c
```