Aztec C65/AS65 Assembler Notes for the Aztec64 Distribution

Prepared by Bill Buckels, August 7, 2013_

Disclaimer and Conditions of Use

The material presented in this document comes with no warranty or guarantee of fitness of use of any kind. It is provided as-is for information purposes only.

The contents of this document and any resulting derivative work may be used for whatever you wish as long as you agree that neither Bill Buckels or the other copyright holders have any warranty or liability obligations whatsoever from said use.

Dedication

The Aztec64 distribution of Aztec C available from the Aztec C Museum **http://www.aztecmuseum.ca** is dedicated to my old friend, colleague and long-time mentor, Leslie Eugene Gros, and this document is dedicated to my new friend Daniel Strang for his dedication to the Aztec64 distribution.

Thanks also to Jeff Hurlbert (Rubywand) for the Manx Aztec C Mini-manual for Apple DOS 3.3 from which some of this material was edited, and to the late Paul R. Santa-Maria for proving the PDF Manual excerpts that appear at the end of this document.

Table of Contents

Forward

This document provides an overview of Assembly Language in the Aztec64 Commodore 64 Aztec C65 cross-compiler distribution available from **http://www.aztecmuseum.ca** (the Aztec C Website).

I created it for Daniel Strang (a fellow Commodore 64 enthusiast) but it may prove useful for others who wish to understand or use assembler in an old Aztec C program for the Commodore 64. Some of this is also applicable to Aztec C65 for the Apple II as well, and even for other old compilers and assemblers.

I no longer have the manual that came with this compiler so the programmer must follow the code that I have provided with Aztec64 for practical examples:

Assembly Source Examples

Several assembly language sources are in Aztec64. Some are listed below:

Original Compiler Assembly Sources
A65 source – hand-written assembler files that end with the .A65 extension:
See Appendix A65 for a listing of these

Additional Assembly Sources
ASM source – compiler generated assembler files
See Appendix ASM for a listing of these

I have also provided the **Aztec C Assembly language portions** from the manual of a later version of Aztec C65 for the Apple II at the end of this section of the document for reference. Much of the information from the newer manual applies to Aztec64's assembler, AS65 because the newer assembler descended from the older AS65.

Do I need To Use Assembler?

No. The compiler does that for you. But if you decide to use assembler you need to be intimate with the Computer and the Compiler at the very lowest levels, and you need to know your 6502 assembly language:

https://en.wikipedia.org/wiki/MOS_Technology_6502

http://homepage.ntlworld.com/cyborgsystems/CS_Main/6502/6502.htm

You also need to know what works and what doesn't, and all I can suggest is that you study and experiment, and learn by trial and error. Back then this was the way that most of us learned assembler, so you will enjoy a vintage learning experience if you decide to go this route… but it will be a very time consuming learning-curve so be prepared to work very long and hard with few results at times.

Assembler Notes for Aztec C65 - Aztec64 distribution

Aztec C65 is a two-pass compiler. In the old days to save memory and compiling time compilers generally compiled to assembly language and then some assembler assembled the assembly language generated by the compiler. Sometimes the programmer "tweaked" the assembly language to make it work better before assembling.

How Things Work

In the Aztec64 Distribution, every Aztec C65 program is compiled from C to assembler first (to an ASM file), then assembled to an Aztec C65 object file (a REL file) and finally linked together to create a binary program image with its load address (org, base address) at $810 in the Commodore 64's memory. The MKBASIC program is then run which appends a small launcher program (a C64 BASIC program) to the front of the binary image file. The program is then finished and ready to be put onto a C64 disk. If it is the first program on the C64 disk it will automatically launch and run.

So therefore every Aztec C program calls assembler routines because every Aztec C program is compiled to assembler first and then assembled. Review any MAKE file in Aztec64 to see how this is done.

What Things Are

The programs used to do so are in the BIN directory.

1. **C65 - compiles C to Assembler**.
2. **AS65 - assembles ASM to REL object files.**
3. **LN65 - links REL object files with other REL files and REL files in LIB files.**
4. MKBASIC - appends BASIC launcher to finished BINary program.

Why Things Are and How They Work

Most C compilers in the 80's and 90's could be used as a 2 pass compiler. Even today most provide an option to produce an assembler listing.

It is seldom if ever necessary to use assembler in a C program if you can do the same thing in C. The compiler takes care of details like saving and restoring the stack, and pusing and popping the stack, and other details that an assembly language programmer would need to do to replace all the C code in a program with assembly language.

However, every compiled C program for older computers must call ROM routines built-in to the hardware of a computer, and if a DOS (Disk-Operating System) is used or special hardware is used, calls to routines in RAM (random access memory) may also be needed. On older computers these calls are done in pure assembler modules that are generally provided as part of the C runtime library that came with the C compiler. Modern C compilers for systems like Microsoft Windows do not generally call ROM or RAM routines directly and instead make calls to an intermediate layer that Windows calls an API (Application Program Interface). While Assembler can be used to make those calls this is generally not done in an application program. On newer computers, device drivers are generally available to avoid the need for direct assembly language calls to hardware ROM routines.

If it is necessary to call a routine in ROM or RAM directly from assembler in a C program, the programmer needs to know how to call and return safely from assembler. Every old compiler did this differently.

What if Assembly is Necessary or Preferable?

It may be necessary or preferable to use assembly language instead of C in the following situations:

1. If directly calling ROM or RAM routines (routines outside your program). Function pointers in C can also be used to do this, but sometimes ROM routines require registers to be loaded and flags to be set before making calls to them. Sometimes register or flag values must be preserved and returned to the C program, so they must be moved to where the C program expects them in memory. In the old days, each hardware company had different ways of doing things so standards developed slowly if at all. The C language is a high level language and provides no standard for dealing with processor registers and other assembler directly. The assembler for the processor does that! So in situations where it is desirable to code at the processor level assembly language is used.

2. C function calls sometimes make extensive use of the stack and the heap especially making repetitive calls in a loop. This can result in much time wasted for time sensitive operations. Assembler can be used to optimize these portions of a program instead of C.

One technique that was common in the old days was to write a SINGLE function in the C language and compile it to assembler, then to optimize the assembler by hand. This provides the C programmer with a template. If you want to use this technique your SINGLE function should be written with as few loops and variables as possible. Do not be afraid to use GOTOs and LABELs.

Generating a commented Assembler (ASM) File with C65

i.e. C65 -T TESTRAND.C

When the -T option is used with the Aztec C65 compiler, the compiler will generate a merged Assembler (ASM) file with the C Program included as comments (preceded with an asterisk *).

This is useful for following the assembly language in the source file, especially for optimizing. Much optimization can be done in C by trying-out different techniques and reviewing each to find the most efficient assembler before even optimizing the assembler.

ZERO page

6502 Computers like the Commodre 64 and the Apple II use ZERO page extensively to interface assembly language to the operating system and the operating system can also use ZERO page. ZERO page is the 256 byte page starting at absolute RAM address of 0.

Aztec C65 uses zero page extensively. Review the C:\Aztec64\INCLUDE\ZPAGE.H file for details.

Inline Assembly

http://en.wikipedia.org/wiki/Inline_assembler

In Aztec C65 you can insert assembly language in a C program and create entire functions or just an assembly line or two. The manual that is attached at the end of this section provides an explanation of how inline assembly works in Aztec C. Even until recently, many compilers provided inline assembly in various formats. I don't know if Aztec C was first or even among the first, but they were the first C compiler I used with a sophisticated inline assembly interface. Microsoft C didn't until the late '80's following Turbo C's lead if I recall correctly. Some compilers also provided other ways to insert inline assembly into C programs, using byte strings and so forth which looked similar to cc65's __asm__ (""); which is ok for a line or two of safe calls…

If you use a line or two of inline assembly in your C programs, make sure your calls don't crash your program. Also be aware that haywire calls can jump to undesired code that could do crazy things like wipe-out disks so experiment on a work disk and make sure you test thoroughly before sending copies of work to other folks. (This is a good practice to follow with any programming of course, but with low level code the risks can be greater especially during learning things like assembler and the use of pointers in C.)

Safety Play – Function Wrappers

One technique that I use as a "safety play" is to wrap inline assembly in a C function body with the hope that the Aztec C65 program will reset everything for me (from B64NAT.LIB's DLIST.C):

```
/* yes virginia, there is inline assembly in Aztec C */
_dlode()
{
#asm
* rather than muck with passing stack args
* i kept it simple and hard-coded the relocatable
* address for the dir buffer which is $1800
LOAD equ $ffd5

* SET FLAG FOR A LOAD
    lda #0
* ALTERNATE START
    ldx #0
    ldy #$18
    jsr LOAD
    rts
#endasm
```

Another technique that I use is to pass parameters to assembly from C on ZERO page using pointers in my C program to load ZERO page before making my assembly call in the wrapped assembler function.

An example of what I mean is shown below. This is Apple II Code, not Commodore 64 code:

```
#asm
instxt    <zpage.h>
COLOR equ REGS
#endasm
unsigned char *byteregptr = (unsigned char *)0x80;
setcolor(value)
{
  /* load parameters into user reg */
  byteregptr[COLOREG] = value;
  /* make ml call */
#asm
  LDA COLOR ;  Sets the plotting color to N, 0 <= N <= 15
  JSR $F864
#endasm
}
```

You can include an entire inline assembly function in your C65 program. But if you are thinking about doing that, why not just keep it in its own module and link it later? The OV program example in Aztec64 shows how that is done.

Closing Remarks

There is much more to say on this whole business of assembly in Aztec C65, but as I said at the beginning, this is an Overview. I hope that this information is useful and good luck with exploring this further.

Appendices and Manual Excerpts follow.

Best Regards,

Bill Buckels
bbuckels@mts.net

Appendix A65 – Hand Written Assembler

The following are Aztec64 run-time library sources and are distributed with Aztec64:

C:\Aztec64\OBJ\BRK.A65
C:\Aztec64\OBJ\INTER.A65
C:\Aztec64\OBJ\OV65.A65
C:\Aztec64\SAMPLES\OV\OV65.A65
C:\Aztec64\SRC\FLTSRC\ATOF.A65
C:\Aztec64\SRC\FLTSRC\CRT1.A65
C:\Aztec64\SRC\FLTSRC\FLT1.A65
C:\Aztec64\SRC\FLTSRC\FLT2.A65
C:\Aztec64\SRC\FLTSRC\FLT65.A65
C:\Aztec64\SRC\FLTSRC\FTOA.A65
C:\Aztec64\SRC\FLTSRC\MATH.A65
C:\Aztec64\SRC\NATIVE\FSTSWT.A65
C:\Aztec64\SRC\NATIVE\ISTACK.A65
C:\Aztec64\SRC\NATIVE\LMATH.A65
C:\Aztec64\SRC\NATIVE\LSHIFT.A65
C:\Aztec64\SRC\NATIVE\MATH.A65
C:\Aztec64\SRC\NATIVE\MOVE.A65
C:\Aztec64\SRC\NATIVE\SHIFT.A65
C:\Aztec64\SRC\NATIVE\STACK.A65
C:\Aztec64\SRC\NATIVE\SUP.A65
C:\Aztec64\SRC\NATIVE\SWIT.A65
C:\Aztec64\SRC\NATIVE\TMPSAV.A65
C:\Aztec64\SRC\OVERLAY\OV65.A65
C:\Aztec64\SRC\OVERLAY\OVINT.A65
C:\Aztec64\SRC\STDIO\AGETC.A65
C:\Aztec64\SRC\STDIO\APUTC.A65
C:\Aztec64\SRC\STDIO\GETC.A65
C:\Aztec64\SRC\STDIO\PUTC.A65
C:\Aztec64\SRC\SYSIO\BLOCKMV.A65
C:\Aztec64\SRC\SYSIO\C64CMD.A65
C:\Aztec64\SRC\SYSIO\C64ERR.A65
C:\Aztec64\SRC\SYSIO\C64SUP.A65
C:\Aztec64\SRC\SYSIO\CALLDEV.A65
C:\Aztec64\SRC\SYSIO\CLEAR.A65
C:\Aztec64\SRC\SYSIO\CRT0.A65
C:\Aztec64\SRC\SYSIO\DEVICE.A65
C:\Aztec64\SRC\SYSIO\INDEX.A65
C:\Aztec64\SRC\SYSIO\INTERP.A65
C:\Aztec64\SRC\SYSIO\LONGS.A65
C:\Aztec64\SRC\SYSIO\RINDEX.A65
C:\Aztec64\SRC\SYSIO\STRCAT.A65
C:\Aztec64\SRC\SYSIO\STRCMP.A65

C:\Aztec64\SRC\SYSIO\STRCPY.A65
C:\Aztec64\SRC\SYSIO\STRLEN.A65
C:\Aztec64\SRC\SYSIO\STRNCMP.A65
C:\Aztec64\SRC\SYSIO\STRNCPY.A65
C:\Aztec64\SRC\SYSIO_EXIT.A65
C:\Aztec64\SRC\XFER14C\C64.A65
C:\Aztec64\SRC\XFER14C\OC64.A65
C:\Aztec64\SRC\XFER14C\SUPERSER.A65

Appendix ASM – Compiler Written Assembler

The following are my own library sources and are distributed with Aztec64:

C:\Aztec64\B64NAT\ASM\ATTEXT.ASM
C:\Aztec64\B64NAT\ASM\BLOAD.ASM
C:\Aztec64\B64NAT\ASM\BOX.ASM
C:\Aztec64\B64NAT\ASM\BSAVE.ASM
C:\Aztec64\B64NAT\ASM\CIRCLE.ASM
C:\Aztec64\B64NAT\ASM\CRTMODE.ASM
C:\Aztec64\B64NAT\ASM\DECODES.ASM
C:\Aztec64\B64NAT\ASM\DISK.ASM
C:\Aztec64\B64NAT\ASM\DLIST.ASM
C:\Aztec64\B64NAT\ASM\FBOX.ASM
C:\Aztec64\B64NAT\ASM\FCHAR.ASM
C:\Aztec64\B64NAT\ASM\GCLR.ASM
C:\Aztec64\B64NAT\ASM\GETCH.ASM
C:\Aztec64\B64NAT\ASM\GETPIXEL.ASM
C:\Aztec64\B64NAT\ASM\KBFLUSH.ASM
C:\Aztec64\B64NAT\ASM\KBHIT.ASM
C:\Aztec64\B64NAT\ASM\LINE.ASM
C:\Aztec64\B64NAT\ASM\MEMSCR.ASM
C:\Aztec64\B64NAT\ASM\OUTTEXT.ASM
C:\Aztec64\B64NAT\ASM\PALSET.ASM
C:\Aztec64\B64NAT\ASM\PFONT.ASM
C:\Aztec64\B64NAT\ASM\PICLODE.ASM
C:\Aztec64\B64NAT\ASM\PLOT.ASM
C:\Aztec64\B64NAT\ASM\PUTIMAGE.ASM
C:\Aztec64\B64NAT\ASM\RB.ASM
C:\Aztec64\B64NAT\ASM\REVTEXT.ASM
C:\Aztec64\B64NAT\ASM\SCR.ASM
C:\Aztec64\B64NAT\ASM\SETLOGO.ASM
C:\Aztec64\B64NAT\ASM\SID.ASM
C:\Aztec64\B64NAT\ASM\XCHAR.ASM
C:\Aztec64\B64NAT\ASM\XCOLOR.ASM
C:\Aztec64\B64NAT\ASM\XFLOAD.ASM
C:\Aztec64\B64NAT\ASM\XSTR.ASM

Appendix AS65 - AS65 6502 Assembler

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

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. 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 ".reI". 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.

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 O 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 Directives

The AZTEC AS65 assembler supports the following pseudo operations:

```
COMMON block name     -sets the location to the selected common block.
CSEG                  -select code segment.
DSEG                  -select data segment
END                   -end of assembler source statements.
ENTRY  expr           -entry point of final module.
EQU  expr             -define label value.
FCB  expr             -define byte constant
FCC  /expr/           -define byte string constant
FDB  expr             -define double byte constant
FUNC label            -if label is not defined then it is declared
external.
INSTXT /file/         -the specified file is included at this point
PUBLIC label          -declares label to be external.
RMB  expr             -reserves expr bytes of memory with no particular
value.
WEAK  expr            -define label value if not previously defined
```

Appendix C65 – C65 Native Code Compiler

c65 [-bts] [-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 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 zero page as work space and temporary registers (defined in ZPAGE.H). C65 also uses 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 "name.asm". Then the AS65 assembler can be used to assemble the "name.asm" file and produce a relocatable object file called "name.rel".

Options

The options available with C65 are listed below.

-o

This option allows the user to specify the name of the output file. This can be used to specify the name of the assembly language file as in:

c65 -o temp.a65 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 ic: 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 C Programming Language

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. The user should refer to that document for a description and definition of the C language.

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

The library routines defined in standard C that are supported by AZTEC C are identical in syntax to the standard. AZTEC C includes some extended library routines that do not exist in the standard C to allow access to native operating system functions. 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.

Appendix LN65 - The Linker

Linking with the Libraries

The LN65 assembler translates 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.

For example, assume that a program consists of two modules, "main.rel" and "subs. reI". 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 bcginning 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 re-locatable object format and a program to link object modules together, the Aztec linker, LN65.

LN65 combines any number of object modules together and produces a binary file in the standard Commodore 64 "BIN" format. LN65 will also indicate if anything is missing.

For this example type:

ln args.rel

In this case, LN65 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.

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, LN65, 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 sa65.lib -b810

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 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 in the current folder. LN65 will call the output file the same name as the first object module argument To specify a different name, LN65 can be used with a "-o" option as follows:

ln -o testprog args.rel sa65.lib -b810

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

And that's all there is to it!

Aztec C65 Manual Excerpt Notes

There is currently no original manual available for this version.

The preceding descriptions of the assembler, compiler, and linker are rewritten manual excerpts from the Apple II C65 native mode compiler of the same vintage. To the best of my knowledge and verification, that information has been accurately rewritten for the Aztec64 distribution.

The section that follows is excerpted from the Apple II Aztec65 version 3.2b manual. Some information may or may not apply to the AS65 assembler in Aztec64, but like the rest of the material in this document it is the best that I can provide at this time.

End of Notes

Manual Excerpts

NEXT PAGE

way to use this parameter would be to declare it to be of type *int*. The user's routine could then convert it to a character string for printing in an error message.

The example below demonstrates how floating point errors can be trapped and reported. In *main*, a pointer in the *Sysvec* array is set to the routine, *usertrap*. If a floating point exception occurs during the execution of the program, this routine is called with the arguments described above. The error handling routine prints the appropriate error message, and returns to the floating point support routines.

```
#include <stdio.h>

main() {
    Sysvec[FLT_FAULT] = usertrap;
}

usertrap(errcode,addr)
int errcode,addr;
{
    char buff[4];

    switch (errcode) {
        case '1':
            printf("floating point underflow at %x\n",buff);
            break;
        case '2':
            printf("floating point overflow at %x\n",buff);
            break;
        case '3':
            printf("division by zero at %x\n", buff);
            break;
        default:
            printf("usertrap: invalid code %d \n", errcode);
            break;
    }
```

3.12 Register Variables

A *cc*-compiled program can have up to eight register variables. A *cci*-compiled program can declare variables to be of type *register*, but the compiler will ignore the declaration.

3.13 In-Line Assembly Language Code

Assembly language source can be included in a C program, by surrounding the assembly language code with the preprocessor directives *#asm* and *#endasm*.

When the compiler encounters a *#asm* statement, it copies lines from the C source file to the assembly language file that it's generating, until it finds a *#endasm* statement. The *#asm* and *#endasm* statements are not copied.

While the compiler is copying assembly language source, it doesn't try to process or interpret the lines that it reads. In particular, it won't perform macro substitution.

A program that uses *#asm ...#endasm* must avoid the following placing in-line assembly code immediately following an *if* block; that is, it should avoid the following code:

```
if (...){
   ...
}
#asm
  ...
 #endasm
  ...
```

The code generated by the compiler will test the condition and if false branch to the statement following the *#endasm* instead of to the beginning of the assembly language code. To have the compiler generate code that will branch to the beginning of the assembly language code, you must include a null statement between the end of the *if* block and the *asm* statement:

```
if (...){
   ...
}
;
#asm
  ...
 #endasm
  ...
```

3.14 Writing machine-independent code

The Aztec family of C compilers are almost entirely compatible. The degree of compatibility of the Aztec C compilers with v7 C, system 3 C, system 5 C, and XENIX C is also extremely high. There are, however, some differences. The following paragraphs discuss things you should be aware of when writing C programs that will run in a variety of environments.

If you want to write C programs that will run on different machines, don't use bit fields or enumerated data types, and don't pass structures between functions. Some compilers support these features, and some don't.

3.14.1 Compatibility Between Aztec Products

Within releases, code can be easily moved from one implementation of Aztec C to another. Where release numbers differ (i.e. 1.06 and 2.0) code is upward compatible, but some changes may be needed to move code down to a lower numbered release. The

downward compatibility problems can be eliminated by not using new features of the higher numbered releases.

3.14.2 Sign Extension For Character Variables

If the declaration of a *char* variable doesn't specify whether the variable is signed or unsigned, the code generated for some machines assumes that the variable is signed and others that it's unsigned. For example, none of the 8 bit implementations of Aztec C sign extend characters used in arithmetic computations, whereas all 16 bit implementations do sign extend characters. This incompatibility can be corrected by declaring characters used in arithmetic computations as unsigned, or by AND'ing characters used in arithmetic expressions with 255 (0xff). For instance:

```
char a=129;
int b;
b = (a & 0xff) * 21;
```

3.14.3 The MPU... symbols

To simplify the task of writing programs that must have some system dependent code, each of the Aztec C compilers defines a symbol which identifies the machine on which the compiler-generated code will run. These symbols, and their corresponding processors, are:

symbol	*processor*
MPU68000	68000
MPU8086	8086/8088
MPU80186	80186/80286
MPU6502	6502
MPU8080	8080
MPUZ80	Z80
MPUINT	Interpreter

Only one of these symbols will be defined for a particular compiler.

For example, the following program fragment contains several machine-dependent blocks of code. When the program is compiled for execution on a particular processor, just one of these blocks will be compiled: the one containing code for that processor.

```
#ifdef MPU68000
   /* 68000 code */
#else
#ifdef MPU8086
   /* 8086 code */
#else
#ifdef MPU8080
   /* 8080 code */
#endif
#endif
#endif
```

THE ASSEMBLERS

Chapter Contents

The Assemblers

as and *asi* are relocating assemblers that translate an assembly language source program into relocatable object code. The two assemblers support different machines: *as* accepts assembly language for a 6502 or 65c02; *asi* accepts assembly language for a "pseudo machine".

In an executable program, an *asi*-assembled module must be interpreted by a routine that is in the Aztec libraries.

An executable program can contain both modules that have been assembled with *as* and modules that have been assembled with *asi*.

This description has three sections: the first describes how to operate the assembler; the second describes the assembler's options; and the third presents information of interest to those writing assembly language programs.

1. Operating Instructions

Operationally, the two assemblers are very similar. In the following paragraphs, we will use the name *as* when referring to features that are common to both assemblers. When the two assemblers differ, we will say so.

as is started with a command line of the form

```
as [-options] prog.asm
```

where *[-options]* are optional parameters and *prog.asm* is the name of the file to be assembled. *as* reads the source code from the specified file, translates it into object code, and writes the object code to another file.

1.1 The Source File

The extension on the source file name is optional. If not specified, it's assumed to be *.asm*. For example, with the following command, the compiler will assume that the file name is *test.asm*:

```
as test
```

as will append *.asm* to the source file name only if it doesn't find a period in the file name. So if the name of the source file really doesn't have an extension, you must compile it like this:

```
as filename.
```

The period tells the assembler not to append *.asm* to the name.

1.2 The Object File

By default, the name of the file to which *as* writes object code is derived from the name of the source code file, by changing its extension to *.o* (or to *.i*, if *asi* is used). Also by default, the object code file is placed in the directory that contains the source code file. For example, the command

```
as test.asm
```

writes object code to the file *test.o* (or to *test.i*, if *asi* is used), placing this file in the current directory.

You can explicitly specify the name of the object code file, using the *-O* option. The name of the object code file follows the *-O*, with spaces between the *-O* and the file name. For example, the following command assembles *test.asm*, writing the object code to the file *prog.out*:

```
as -o prog.out test.asm
```

1.3 The Listing File

The *-L* option causes the assembler to create a file containing a listing of the program being assembled. The file is placed in the directory that contains the object file; its name is derived from that of the object file by changing the extension to *.lst*.

1.4 Searching for *instxt* files

The *instxt* directive tells *as* to suspend assembly of one file and assemble another; when assembly of the second file is completed, assembly of the first continues.

You can make the assembler search for *instxt* files in a sequence of directories, thus allowing source files and *instxt* files to be in different directories.

Directories that are to be searched are defined just as for the compilers; that is, using the -I assembler option and the INCLUDE environment variable. Optionally, the compiler can also search the current directory.

Directory search for a particular *instxt* directive can be disabled by specifying a directory name in the directive. In this case, just the specified directory is searched.

1.4.1 The -I option

A -I option defines a single directory to be searched. The directory name follows the -I, with no intervening blanks. For example, the following *-I* option tells the assembler to search the *include* directory on the *ram* volume:

-I/ram/include

1.4.2 The INCLUDE environment variable.

The INCLUDE environment variable defines a directory to be searched for *instxt* files. For example, the following command sets *INCLUDE* so that the compiler will search for *instxt* files in the directory */ram/include*:

set INCLUDE=/ram/include

See the SHELL chapter for details on the setting of environment variables.

1.4.3 The search order

Directories are searched in the following order:

1. If the *instxt* directive delimited the file name with the double quote character, ", the current directory on the default drive is searched. If delimited by angle brackets, < and >, this directory isn't automatically searched.
2. The directories defined in -I options are searched, in the order listed on the command line.
3. The directory defined in the INCLUDE environment variable is searched.

2. Assembler Options

The assembler supports the following options:

Option	*Meaning*
-O objname	Send object code to *objname*.
-L	Generate listing.
-C	Disable assembly of 65C02 instructions. Not supported by *asi*.
-ZAP	Delete the source file after assembling it.

3. Programming Information

This section discusses the assembly language that is supported by *as*. A description of the assembly language supported by *asi* is not available.

as supports the standard MOS Technology syntax: a program consists of sequence of statements, each of which is in the standard MOS Tech form; and the assembler supports the MOS Tech mnemonics for the standard instructions. *as* supports some of the MOS Tech directives and their mnemonics; it also supports others, as defined below.

The following paragraphs define in more detail the language supported by *as*.

3.1 Statement Syntax

[label] [opcode] [arguments] [[;]comment]

where the brackets "[...]" indicate an optional element.

3.2 Labels

A statement's label field defines a symbol to the assembler and assigns it a value. If present, the symbol name begins in column one. If a statement is not labeled, then column one must be a blank, tab, or asterisk. An asterisk denotes a comment line.

Normally, the symbol in a label field is assigned as its value the address at which the statement's code will be placed. However, the *equ* directive can be used to create a symbol and assign it some other value, such as a constant.

A label can contain up to 32 characters. Its first character must be an alphabetic character or one of the special characters '_' or '.'. Its other characters can be alphabetic characters, digits, '_', or '.'. A label followed by "#" is declared external.

The *cc* compiler places a '_' character at the end of all labels that it generates.

3.3 Opcodes

The assembler supports the standard MOS Tech instruction mnemonics for both the 6502 and 65C02 processors. The directives it supports are defined below.

3.4 Arguments

A statement's arguments can specify a register, a memory location, or a constant.

A memory location can be referenced using any of the standard 6502 or 65C02 addressing modes, and using the standard MOS Tech syntax.

A memory location reference or a constant can be an expression containing any of the following operators:

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

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

3.5 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 consists of the character, preceded by a single quote. For example: 'A.

3.6 Directives

The following paragraphs describe the directives that are supported by the assembler.

END

end

The *end* directive defines the end of the source statements.

CSEG

cseg

The *cseg* directive selects a module's code segment: information generated by statements that follow a *cseg* directive is placed in the module's code segment, until another segment-selection directive is encountered.

DSEG

dseg

The *dseg* directive selects a module's data segment: information generated by statements that follow a *dseg* directive is placed in the module's data segment, until another segment-selection directive is encountered.

EQU

symbol equ <expr>

The *equ* directive creates a symbol named *symbol* (if it doesn't already exist), and assigns it the value of the expression *expr.*

PUBLIC

public <symbol>[,<symbol>...]

The *public* directive identifies the specified symbols as having external scope. If a specified symbol was created within the module that's being assembled (by being defined in a statement's label field), this directive allows it to be accessed by other

modules. If a symbol was not created within the module that's being assembled, this directive tells the assembler that the symbol was created and made public in another module.

BSS

bss *<symname>,<size>*

The *bss* directive creates a symbol named *symnam* and reserves *size* bytes of space for it in the uninitialized data segment. The symbol cannot be accessed by other modules.

GLOBAL

global *<symnam>,<size>*

The *global* directive creates a symbol named *symnam* that other modules can access using the *global* and *public* directives.

If other modules create *symnam* using just the *global* directives, then *symnam* will be located in a program's uninitialized data area. In this case, the amount of space reserved in this area for *symnam* will equal the largest value specified by the *size* fields in the *global* statements that define *symnam*.

If other modules define *symnam* in a *public* statement, but none of them create *symnam* (by specifying it in a label field), then *symnam* will still be located in the uninitialized data segment and space will be reserved for it as defined above.

If one module both defines *symnam* using a *public* statement and creates the symbol by specifying it in a label field, then *symnam* will be located in the program's code or data segment and no space will be reserved for it in the uninitialized data segment.

ENTRY

entry *<symnam>*

The *entry* directive defines the symbol *symnam* as being a program's entry point.

When a program is linked, the linker normally places a jump instruction at the program's base address. If the linker found a module containing an *entry* directive, it sets the target of the jump to the location that was specified in the last *entry* directive that it found; otherwise, it sets the target to the beginning of the program's code segment.

FCB

[label] *fcb* *<value>[,<value>, <value> ...]*

Each *value* in an *fcb* directive causes one or more bytes of memory to be allocated and then initialized to the specified value. The memory is allocated in the currently active segment

(code or data, as defined by the last segment-selection directive).

FDB

[label] *fdb* *<value>[,<value>, <value> ...]*

The *fdb* directive is like *fcb*, except that each *value* causes a two-byte field of memory to be allocated and initialized.

FCC

[label] *fcc* *"string"*

The *fcc* directive allocates a field that has the same number of characters as are in *string*, and places *string* in it. The field is placed in the currently-active segment.

RMB

[label] *rmb* *<expr>*

The *rmb* directive reserves a field containing *expr* bytes in the currently-active segment. The contents of the field are not defined.

INSTXT

instxt *<file>*
instxt *"file"*
instxt */file/*

The *instxt* directive causes the assembler to suspend assembly of the current source file and to assemble the source that's in *file*. When done, the assembler will continue assembling the original file.

The assembler can search for a file in several directories. If *file* is surrounded by quotes or by slashes, the assembler will begin the search at the current directory; it will then search directories specified in the -I option and the INCLUDE environment variable. If *file* is surrounded by <>, the assembler will search just the -I and INCLUDE directories.