AZTEC C ...the most portable and comprehensive C software development system available

Manx Software Systems with over 6,000 licensed users is one of the leading suppliers of "C" compilers and cross compilers for the world's most popular micro computers. The MANX AZTEC C compiler is available as a cross compiler or native compiler for the following environments:

AZTEC C86	AZTEC C II	AZTEC C65
PC DOS	CP/M-80	APPLE DOS
MS DOS	TRSDOS	COMMODORE 64 (12/83)
CP/M-86		ProDOS (2/84)

The AZTEC C product is a complete development system. In addition to a full v7 "C" compiler, the basic product includes assemblers, linkage editors, development utilities, and full run time libraries. New products to be released in late 1983 and early 1984 include graphics development tools, data base managers, program editors, screen management systems, and other development tools that work in conjunction with the AZTEC C system.

CROSS DEVELOPMENT SYSTEMS

The AZTEC C cross development systems include all of the utilities and library support routines available with the native versions including a cross assembler. The binary image created in the host environment is downloaded and tested in the target environment. MANX has been using its own cross compilers on a daily basis since 1980.

HOSTS

PDP-11	68000	8086/8088	8080/8085/Z80
UNIX	UNIX ports	UNIX ports	CP/M-80
		PC DOS	
		MS DOS	
		CP/M-86	

TARGETS

8086/8088	8080/8085/Z80	6502/65xx
PC DOS	CP/M-80	APPLE DOS
MS DOS	TRSDOS (12/83)	APPLE ProDOS (2/84)
CP/M-86	LDOS (12/83)	COMMODORE 64
	DOSPLUS (12/83)	

Other host and target environments will be released in early 1984.

NATIVE COMPILERS

The AZTEC C native 8080 compiler was first released in 1981. Since that time it has been acquired by more than 300 colleges and universities, thousands of corporations, small business, and government agencies. The compiler has been ported to the 6502 and 8086. Plans for future ports include the 68000, 16032, and IBM 370. All native versions are source compatible. Source developed in one environment can therefore be transferred to another environment and compiled, linked and executed.

For prices and information call:
***800-221-0440** (outside NJ)*
201-780-4004 (inside NJ)

Or write to: ***MANX SOFTWARE SYSTEMS***
P.O. BOX 55 • SHREWSBURY, NJ 07701

MANX®
software systems

Aztec C for CPM, version 1.06D
Release Document

This release document describes Aztec C for CPM, version 1.06D, and is divided into the following sections:

1. Differences between Aztec C, versions 1.06D and 1.06B
2. Differences between Aztec C, versions 1.06B and 1.05G
3. Packaging
4. Helpful hints
5. Outstanding bugs
6. Addenda

1. Differences between Aztec C, versions 1.06D and 1.06B

Version 1.06D of Aztec C for CPM is primarily a bug-fixing release. There are a few additions to the compiler preprocessor.

1.1 Fixed bugs

- All Manx programs, except for the compilers, cc and cz, now recognize when there is no more disk space, and log an error message.

- The linker had a bug which caused some overlays to overlay the last byte in the root's uninitialized data segment.

- The compilers had a bug which caused them to ignore the line following a #endasm statement.

- There was a bug in the function agetc() which prevented a program from turning off the EOF flag on the console.

- The format() function, called by printf(), fprintf(), and sprintf(), incorrectly handled the '%*f' conversion.

- The compilers incorrectly generated an error 99 for statements in which a character pointer was assigned to a long variable.

- The compilers generated incorrect code for statements of the form

```
a <= 4;
a /= 4;
```

  when 'a' was a character variable.

- Some bugs in the ftoa() and atof() function were fixed.

- The internal function for closing a file, filecl(), didn't properly handle files located in user areas other than the current area. This prevented programs from accessing files in other user areas.

- The program 'sidsym' will access files in any user area, and will sort a symbol table into numerical order. It has been linked with the tiny library, thus reducing its size.

- The 1.06D version of the linker required that the extension '.com' for the file to which executable code was written. If another extension was used, the linker would generate inappropriate error messages. With the 1.06D linker, if the extension isn't '.com', the linker will generate a memory image of the executable program; in this case, the default base address and code segment address are 0.

- The functions in() and out() have been added to c.lib.

- The module 'ctype' was misplaced in the library: it now comes after the modules 'atoi' and 'atol', since they both reference it.

- There was a bug in the tiny library, t.lib, which affected programs that called puts() without calling putchar() or that called gets() without calling getchar(). This has been fixed by adding getchar() and putchar() to the Croot module in t.lib.

- There was a bug in the compilers which occurred when both the -M and -U options were used in the compilation of a program: global functions declared in the program weren't declared 'public' and hence couldn't be accessed by functions in other modules.

- The rename() function now returns -1 when it fails.

- The 'exec' functions were fixed.

- There was a bug in code that compared a long variable to a constant.

- The compiler preprocessor didn't support \" and \'.

- The compiler generated incorrect code when the ++ operator was applied to an unsigned long variable.

- There was a bug in malloc() which sometimes caused programs calling it to go into an infinite loop.

- There was a bug in the processing of #if statements by the compiler which affected statements of the form

```
#if MACRO
```

  when MACRO wasn't defined.

1.2 New features

- The compilers now have the ability to print error messages instead of an error code. The file 'cc.msg' contains the error messages. If the compiler finds an error, and if it can open this file, the error's message will be printed; otherwise, the error's number will be printed.

  The compiler searches for cc.msg in the same areas that it would search for an include file.

- The Aztec C compilers now have predefined symbols which identify the machine on which code generated by a compiler will run:

```
MPU8080   -    code will run on an 8080
MPUZ80    -    code will run on a Z80
MPU8086   -    code will run on an 8086 or 8088
```

This allows statements of the form

```
#if MPU8080 | MPUZ80
     /* 8080 and Z80 code goes here */
#else
#ifdef MPU8086
     /* 8086 code goes here
#endif
#endif
```

- The compiler now treats adjacent quoted character strings as a single quoted string. For example, the statement

```
printf("this is" " really just" "one string");
```

is equivalent to

```
printf("this is really just one string");
```

- The compiler defines the following symbols to be quoted strings:

__LINE__
: The number of the line being compiled.

__FILE__
: The name of the file being compiled.

__FUNC__
: The name of the function being compiled.

This, and the addition noted above allows statements like:

```
printf("error in file " __FILE__ " line  " __LINE__);
```

- The compiler supports the #line statment. This has the syntax

```
#line lineno filename
```

and resets the compiler's idea of the file being compiled to 'filename' and of the current line number to 'lineno'.

2. Differences between Aztec C, versions 1.06B and 1.05G

This section discusses the changes which were made in going from version 1.05G to 1.06B.

2.1 Compiler changes

The following changes were made to the 8080 and Z80 compilers:

- #if is now supported.

- If the last character on a line is the backslash character, '\', the compiler will consider the next line to be part of the curret line. Thus, lines can be indefinitely long.

- The compiler will allow the file name in a #include statement to be delimited by angle brackets, '<' and '>', as well as by double quotes. That is,

  ```
  #include <filename>
  ```

  is supported.

- The compiler now supports **unsigned char** and **unsigned long** in adition to **unsigned int.** As before, **unsigned** defaults to **unsigned int.** On 8-bit machines, **char** is unsigned; on 16-bit machines, **char** is signed.

- The compiler now supports 'short int' declarations.

- The compiler supports the following new options, which are fully described in the manual:

  ```
  -f   In-line function entry code.
  -i   areas to be searched for include files.
  -l   size of local symbol table.
  -p   send error messages to the printer.
  -q   convert automatic variables to static.
  -r   produce code for RMAC by Digital Research.
  -u   convert globals to externs.
  ```

- The compiler now does some mild type checking, which may cause compilation errors for code which previously compiled without errors.

  Variables cannot be redeclared. The compiler will generate an error in the obvious case:

  ```
  int i;
  double i;
  ```

  Nor can functions be redeclared. Hence the following will produce a compilation error:

```
main()
{
     double func();
     int i;
}

func()
{}
```

func has been declared **double** in **main** but defined as **int.**

2.2 Changes to the linker and libraries

The major change in the linker in going from version 1.05G to 1.06B is that it now distinguishes between initialized and uninitialized data, which are placed in separate regions. The linker has new options accordingly for specifying the address of each segment.

The standard run-time library is now called **c.lib**; the library containing floating point functions is called **m.lib.**

When a program that performs floating point is linked, the floating point library, **m.lib**, must be searched by the linker before the standard library, **c.lib.**

A new library, **t.lib**, is provided which will decrease the size of a program. It's uses and limitations are described in the manual.

Programs can be quickly linked with the new library, **r.lib.** They must be loaded with the program **r.com.** These two files aren't on the distribution disks; instead, a batch file is provided with which they can be created.

2.3 Generating ROM-able code

Version 1.06B of the Aztec C package has more support for generating ROM-able code than did version 1.05G.

A ROM-able program can have pre-initialized data.

A program now has three segments: code, initialized data, and uninitialized data. When a program is started, its uninitialized data segment is automatically cleared.

The compiler now generates public symbols for global variables rather than common blocks. With this, variables can be easily located in ROM.

A special library, rom.lib, is provided. Programs linked

with it are smaller than programs linked with c.lib, since it doesn't automatically pull in the standard UNIX-compatible i/o functions, as does c.lib. Programs linked with it aren't passed command line arguments, and don't have access to the stdin, stdout, and stderr devices.

A utility program, hx, is provided for converting the memory image of program, as generated by the linker, into Intel hex format, for feeding to a ROM burner.

2.4 Function changes

In version 1.06B, some functions have been added, some features added to 1.05 functions, and some 1.05 functions deleted.

2.4.1 New functions

ioctl

With this function, programs can handle console i/o in a variety of ways: console input can be performed a line or character at a time, with or without echo. For a full description, see the console i/o section of the functions chapter in the manual.

setjmp, longjmp

these standard UNIX functions allow a program to escape to a known point when neccessary.

isxxx

These functions, implemented as macros, allow a program to classify characters.

qsort

A sort function.

setmem

Set memory to a specific value.

movmem

Move a block of memory

sbrk

Primitive memory allocation function

malloc, calloc, realloc, free

Sophisticated memory allocation functions.

execl, execv, execlp, execvp

These functions allow a program to activate another program. Control is never returned to the calling program.

2.4.2 Additions to existing functions

- **scanf** has been brought up to the UNIX standard.
- **g** option added to **printf.**

2.4.3 Deleted functions

blockmv	-	use **movmem** instead
settop	-	use **sbrk**
clear	-	use **setmem**

2.5 Fixed bugs

The following bugs were fixed in going from version 1.05G to version 1.06B:

- A bug in the comparison of longs is fixed.
- The function **fopen** when used to open a file in append mode, will now correctly position a file containing text.
- The compiler will work with files which are an exact multiple of 128 bytes in length.
- **open** frees the file control block of a file on an open failure.
- **scanf** works according to the description in the library section of the manual. Several bugs were fixed.
- The logical negation of a constant now works. For example, !1 is zero.
- %% in a format string is treated as the per cent character.
- The initializer of an automatic or register variable can be an arbitrary expression.
- The extract option (-x) for **libutil** has been fixed.
- The **%g** conversion is supported by the **printf, fprintf,** and **sprintf** functions.

- The compiler allows macros to be redefined.

- Under the -M compiler option, the caveats for using M80 no longer exist. This includes: specifying an option to M80 to ensure that statics are initialized to zero; including libc.h in every source module; and specifying the .8080 statement to M80.

3. Packaging

This section describes the files provided with the Aztec C package.

3.1 Standard package

cz.com	Z80 compiler
cc.com	8080 compiler
cc.msg	compiler error message file
as.com	8080 assembler
ln.com	linker
hx.com	Intel hex generator
arcv.com	Source dearchiver
libutil.com	Object file librarian
sidsym.com	Utility for use with DRI SID/ZSID
c.lib	standard run-time library
m.lib	library of floating point functions
t.lib	tiny library
crc.com	crc program
header.arc	archived header files
ovloader.o, ovbgn.o	ovloader support functions
r.o, rbegin.o, rext.asm, rbuild.sub	files for making r.com and r.lib
exmpl.c	sample C source program

3.2 Pro extensions

The following source archive files are provided:

libcsrc.arc	C programs
libasrc.arc	assembler programs
mathsrc.arc	floating point functions
ovly.arc	overlay functions
hx.arc	Intel hex generator
tinysrc.arc	Source for t.lib

Other files in the pro extensions:

rom.lib	library used in generating ROMable code
libc.rel, math.rel	versions of c.lib and m.lib for use with M80 and RMAC
cnm.com	object file utility
sqz.com	object file utility

Source archives contain the source for many separate functions, and can be unpacked into individual files by the program **arcv:**

```
arcv header.arc
```

unpacks all the files in header.arc to separate files on the default drive.

3.3 Checking the files

To verify that the files on the disk are correct, the program 'crc' can be run. This computes a number, called the 'crc', for each specified file. The number generated by a file can be compared to the correct numbers, which are listed below.

The command to start crc has the form

```
crc [filename]
```

If 'filename' isn't specified, the crc is computed for each file on the current user area on the default drive.

'filename' can specify a single file. It can also define a set of files using the standard CPM 'wildcard characters' * and ?. For example,

```
crc *.arc
```

computes the crc of each file having extension '.arc'.

The crc's of the files in the basic package are:

cc.com	49D7	arcv.com	F71C
libutil.com	C4CC	cz.com	9AA6
t.lib	F76C	as.com	492C
ln.com	6B78	c.lib	EBCA
m.lib	8348	rbegin.o	A830
r.o	09F4	header.arc	6618
rbuild.sub	5895	cc.msg	61E3
rext.asm	5441	sidsym.com	5517
exmpl.c	3780	ovbgn.o	1792
ovloader.o	0F29	crc.com	C4CF

The crc's of the pro extension files are:

libc.rel	CC10	libcsrc.arc	F6B8
libasrc.arc	B783	tinysrc.arc	DAF6
mathsrc.arc	ABEF	ovly.arc	F32E
rom.lib	6298	sqz.com	7D54
cnm.com	9990	math.rel	4205

4. Helpful hints

This section discusses common problems encountered when using Aztec C.

- If all the files don't seem to be on your disks, check the reverse side of the disks. Some systems allow information to be read from only one side of a disk. For such systems, we frequently send out disks which have information on both sides. By putting a disk in your drive with one side up you can read the information on that side, and by putting it in the drive with the other side up, you can read the information on the other side.

- If a program performs floating point operations, it must be linked with **m.lib**. The linker must search this library before **c.lib**. That is, the link line must look something like

```
ln prog.o m.lib c.lib
```

  If you have a printf statement with a '%f" conversion, and printf prints '%f' instead of a floating point number, you have specified the libraries to the linker in the wrong order.

5. Outstanding bugs

This section describes bugs which exist in version 1.06D of C.

- The compiler doesn't print an error message when there is no more space on a disk to which it is writing.

6. Addenda

This section presents information which was omitted from the manual.

6.1 Creating the 'fast linker' files

To save disk space, the 'fast linker' files r.com and r.lib are not provided on the distribution disks. Instead, a submit file named **rbuild.sub** is provided, with the files r.o, rext.asm, and rbegin.o, which will build r.com and r.lib.

6.2 RMAC patch

RMAC doesn't allow symbols to contain the characters '.' or '_', both of which are required by Aztec C-supplied programs and by compiled programs.

RMAC can be patched to allow these characters in symbol names. The procedure for doing this is described in pages which are appended to this release document.

6.3 HX and SIDSYM documentation

Descriptions of the programs HX and SIDSYM were omitted from the manual, and are appended to this release document.

NAME

hx - Intel hex generator

SYNOPSIS

hx infile [options]

DESCRIPTION

hx converts the memory image version of a program to Intel hex format. A program which is to be burned into ROM is often required to be in this format.

hx is used in this way:

hx infile [options]

where **infile** is the name of the file, generated by the linker LN, which which contains the memory image of the program. **[options]** are optional parameters which are described below.

hx also reads the symbol table for the program, which must have the same name as **infile**, with the extension .SYM. The option -T causes the linker to generate this file.

An optional period (.) on the command line causes **hx** to send its output to the standard output device, which of course can be redirected to a disk file. Absence of this option causes **hx** to send its output to a file whose name is derived from **infile** by changing the extent to .HEX.

For example, given a program whose memory image and symbol table are in the files PROG.COM and PROG.SYM, the following will generate Intel hex code for it in the file PROG.HEX:

HX PROG.COM

And the following will send the hex code to the file OUTPUT.FIL:

HX PROG.COM . >OUTPUT.FIL

The option **-B** defines the load address for the first record generated by **hx**. It defaults to 0x100. For example, the following will begin loading PROG.COM at 0x8000:

HX PROG.COM -B8000

II. In more detail...

Intel hex code consists of a sequence of 16-byte records, each having the following format:

```
:llaaaattdd..ddccCRLF

  ll = record length (up to sixteen bytes)
aaaa = load address
  tt = record type (0, except for end-of-file)
d..d = data bytes
  cc = checksum (0 - sum of bytes in record)
  CR = carriage return
  LF = linefeed
```

hx generates Intel hex code for a program's code segment and initialized data segment. When this code is burned into ROM, the initialized data segment will immediately follow the code segment in memory. When the code is activated, the Manx routine **.begin**, which initially gets control, will move the ROM copy of the initialized data segment into RAM.

NAME

sidsym - generate SID-readable symbol table

SYNOPSIS

sidsym infile outfile

DESCRIPTION

sidsym converts a symbol table which has been generated by the Manx linker, **ln** to a format which can be read by the Digital Research symbolic debugger program, SID.

The linker option **-T** causes the linker to generate a file containing the symbol table.

infile is the name of the symbol table file created by the linker.

outfile is the name of the file in which **sidsym** is to place the reformatted symbol table. It can have the same name as **infile**

EXAMPLES

The following command will link the object file **exmpl.o**, creating the files **exmpl.com** and **exmpl.sym**, which contain the executable program and the symbol table, respectively:

```
ln -t exmpl.o -lc
```

The following will then convert the symbol table in exmpl.sym to SID-readable format, leaving the result in **exmpl.sym:**

```
sidsym exmpl.sym exmpl.sym
```

Patches for RMAC

When using **RMAC** there are several restrictions on what characters can be used in labels. **RMAC** has several restrictions on legal labels.

- Leading '.' are not allowed in labels. The **CII** run-time library entry points begin with a '.'.
- Imbedded or trailing '_' are not allowed in labels. Many C programs often use '_' in labels and the compiler also appends a trailing '_' character to some labels.

The following patch can be applied to **RMAC** to allow labels containing '.' and '_' characters.

1. First determine your version of **RMAC**

   Enter

   ```
   rmac
   ```

   RMAC will start, list its version number, log a message saying that no source file was entered, and halt.

2. RMAC 1.0 Patch

   RMAC must be patched using the Digital Research program DDT. To start DDT, enter

   ```
   ddt rmac.com
   ```

   DDT will display several lines, then display a '-' character, which is its prompt, and then wait for a command to be entered.

   First enter

   ```
   L1d91
   ```

   followed by a carriage return. DDT will display the RMAC instructions beginning at 0x1d91, the first two of which should be

   ```
   1d91 CPI   3F
   1d93 JZ    1DA6
   1D96 ...
   ```

   Then enter

   ```
   L1 3b
   ```

to display the instruction beginning at 0x13b. The first instruction should be

```
013B NOP
13c  ...
```

Now enter

```
A1d93
```

followed by a carriage return, to patch the instruction at 0x1d93. DDT will display the address and wait for you to enter the new instruction. Enter

```
jmp  13b
<cr>
```

Now enter

```
A13b
```

followed by a carriage return, to patch the instructions beginning at 0x13b. DDT will display the address and wait. Enter

```
jz   1da6
cpi  2e
jz   1da6
cpi  5f
jz   1da6
jmp  1d96
<cr>
```

All the patches have been made. Now exit DDT by entering

```
g0
```

followed by a carriage return. Then save the patched RMAC by entering

```
save 53 crmac.com
```

This saves the patched RMAC in the file 'crmac.com'.

3. RMAC 1.1 Patch

RMAC must be patched using the Digital Research program DDT. To start DDT, enter

```
ddt rmac.com
```

DDT will display several lines, then display a '-' character, which is its prompt, and then wait for a command

to be entered.

First enter

```
L1d9c
```

followed by a carriage return. DDT will display the RMAC instructions beginning at 0x1d9c, the first two of which should be

```
1d9c CPI  3F
1d9e JZ   1DB1
1D96 ...
```

Then enter

```
L13b
```

to display the instruction beginning at 0x13b. The first instruction should be

```
013B NOP
13c  ...
```

Now enter

```
A1d9e
```

followed by a carriage return, to patch the instruction at 0x1d9e. DDT will display the address and wait for you to enter the new instruction. Enter

```
jmp  13b
<cr>
```

Now enter

```
A13b
```

followed by a carriage return, to patch the instructions beginning at 0x13b. DDT will display the address and wait. Enter

```
jz   1db1
cpi  2e
jz   1db1
cpi  5f
jz   1db1
jmp  1da1
<cr>
```

All the patches have been made. Now exit DDT by entering

```
g0
```

followed by a carriage return. Then save the patched RMAC by entering

```
save 53 crmac.com
```

This saves the patched RMAC in the file 'crmac.com'.