SOFTWARE DEVELOPMENT SERIES

MANX™

Aztec C68k/ROM v3.6b, for PCDOS/MSDOS Host Systems
Release Document
24 Jun 1988

This package contains version 3.6b of the Aztec C68k/ROM Cross Development System. It's used to develop, on a PCDOS or MSDOS host system, programs that will run on a 68k ROM-based system.

This release document contains the following sections:

1. Product overview
2. New features
3. Known bugs

1. Product overview

1.1 Components

Aztec C68k/ROM consists of the following components:

* The Aztec C68k/ROM software;
* Aztec C68k/ROM documentation;
* Aztec C documentation, which describes features that are common to all Aztec C packages. This documentation describes the following topics: (a) library functions that are provided with all Aztec C packages; (b) an overview of these functions; (c) a general discussion on the writing of C programs; and (d) compiler error messages.

Aztec C68k/ROM is used in conjunction with either Aztec Host PC, or the Developer or Commercial version of Aztec C86. Aztec C68k/ROM contains programs (such as the compiler, assembler, and linker) that are used to develop 68k/ROM programs, while Aztec Host PC or Aztec C86 contains programs that facilitates development.

The documentation for Aztec C68k/ROM hasn't changed, except as described in this release document. Thus, if you are a current user of Aztec C68k/ROM, this release document is the only documentation in your update.

1.2 Getting started

The Tutorial chapter of the Aztec C68k/ROM manual describes how to install and start using Aztec C68k/ROM.

2. New features

This section describes the that have been added to Aztec C68k/ROM in going from version 3.4b to 3.6b.

2.1 New features of the C68 compiler

The following paragraphs describe features that are new in v3.6b of the *c68* compiler.

2.1.1 Code Generator Control

The Xn option allows special control of the code generator. Each option takes the form of +Xn, where n is a decimal number indicating the option choice. It is possible to abbreviate these options by specifying +X1, 2 ,3 instead of having a separate +X for each option. The extended options currently supported are:

+X1 - Remove A6 from all lists. The compiler considers it not to exist.

+X2 - Places data into code segment.

+X3 - Delays the popping of arguments until it is necessary. This saves some small amount of code when several functions are called one after another. Stack is corrected whenever it is necessary or if the number of bytes delayed exceeds 50 bytes. This affects the Db stack back trace command since DB checks the instruction at the return address to see how many arguments were passed to the function.

+X4 - Forces literal strings to be aligned on an even boundary.

+X5 - If enabled, +X5 generates inline code for strcpy(), strcmp(), and strlen() or if the name used is preceded by __BUILT-IN __STRCPY(S1,S2)

2.2 Source level debugging

The *-n* option causes the compiler to generate source level debugging information. This information is passed as special lines and characters in the assembly language output file. The assembler automatically passes source level debugging information through to an object module.

2.3 Library changes

2.3.1 Math libraries

There are now two math libraries supported by Aztec C68k/ROM:

* *m881.lib*, for programs that perform floating point operations using the 68881 coprocessor.
* *m68.lib*, for programs that perform floating point operations using software emulation.

Source for *m881.lib* is in *m881.arc*. It should be dearchived into a new subdirectory of the *lib* directory, named *m881*.

Source for *m68.lib*, which used to be in an archive named *flt68.arc*, is now in *math.arc*.

*

3. **Known bugs**

* *as68* doesn't support the *.b* and *.w* extensions for *bsr, bra,*, etc. It also doesn't support the *.s* and *.l* extensions for the *jmp* and *jsr* instructions. However, these are generally not necessary, since the assembler generates the correct length instruction.

* *as68* reports errors when the following is assembled with *-c* and *-d* options:

```
        dseg
regsave ds.l    16
        cseg
        public  .begin
        entry   .begin
        movem.l         d0-d7/a0-a7,regsave
        dbra    d0,.2
```

If the code is assembled with *-c*, *-d*, and *-n*, no errors are reported.

* The following code, when compiled with *+f8*, sends *c68* v3.6b into an infinite loop.

```
long a; double b;
main()
{
        a = b++;
}
```

Aztec C68k/ROM v3.4b, for PCDOS/MSDOS Host Systems
Release Document
8 Feb 1988

This package contains version 3.4b of the Aztec C68k/ROM Cross Development System. It's used to develop, on a PCDOS or MSDOS host system, programs that will run on a 68k ROM-based system.

This release document contains the following sections:

1. Product overview
2. New features
3. Known bugs
4. Packaging

1. Product overview

1.1 Components

Aztec C68k/ROM consists of the following components:

* The Aztec C68k/ROM software;
* Aztec C68k/ROM documentation;
* Aztec C documentation, which describes features that are common to all Aztec C packages. This documentation describes the following topics: (a) library functions that are provided with all Aztec C packages; (b) an overview of these functions; (c) a general discussion on the writing of C programs; and (d) compiler error messages.

Aztec C68k/ROM is used in conjunction with either Aztec Host PC, or the Developer or Commercial version of Aztec C86. Aztec C68k/ROM contains programs (such as the compiler, assembler, and linker) that are used to develop 68k/ROM programs, while Aztec Host PC or Aztec C86 contains programs that facilitates development.

The documentation for Aztec C68k/ROM hasn't changed, except as described in this release document. Thus, if you are a current user of Aztec C68k/ROM, this release document is the only documentation in your update.

1.2 Getting started

The Tutorial chapter of the Aztec C68k/ROM manual describes how to install and start using Aztec C68k/ROM. The following paragraphs discuss topics that aren't covered in that chapter; namely, installation on floppy disks.

If your system doesn't have a hard disk, you'll have to use several floppy disks for program development, and swap them in and out of

drives as needed. One possible organization of files on the disks is this:

* On one disk, put *c68*, *as68*, *ln68*, *srec68* or *hex68*, 'include' files, object module libraries. On this disk you could also put a text editor, such as the *Z* text editor that is in both the Aztec Host PC and Aztec C86 packages.
* On another disk, put the less-frequently used programs such as the Aztec C68k/ROM object module utilities, and the *grep* and *diff* utilities that are in Aztec Host PC and Aztec C86.
* On another disk, put the files that are used to create libraries, such as library source, *lb68*, and the *arcv* and *make* utilities that are in Aztec Host PC and Aztec C86. Copy libraries that you create to the first disk.
* On other disks, put your own files.

2. New features

This section describes the that have been added to Aztec C68k/ROM in going from version 3.30c to 3.4b.

2.1 New features of the C68 compiler

The following paragraphs describe features that are new in v3.4b of the *c68* compiler.

2.1.1 68020 support

The +2 option causes the compiler to generate 68020 code.

2.1.2 68881 support

A program's floating point operations can now be performed either by floating point emulation that conforms to the IEEE standard, or by a 68881 coprocessor. When emulation is selected, floating point operations are performed by calling software routines that are in *m68.lib*. When 68881 support is selected, floating point operations are performed in-line as much as possible, and, when necessary, by calling the routines in the new *m881.lib* math library.

Two options have been added to the *c68* compiler, which define how floating point is to be performed.

+fi The *+fi* option, which is the default, selects software emulation. When this option is used, a program must be linked with *m68.lib* (or its large code, large data version, *m68ll.lib*). Up to two register variables may be declared with this format; they reside in register pairs D4-D5 and D6-D7.

+f8 The *+f8* option selects 68881 execution of floating point operations. When this option is used, a program

must be linked with *m881.lib* (or with its large code, large data version, *m881ll.lib*). Four register variables may be declared; they reside in registers FP4-FP7.

2.1.3 Stack depth checking: the +M option

The compiler's new *+m* option causes stack-depth checking to be performed on function entry, by generating a call to the assembly-language function *___stkchk*. If *__stkchk()* detects that the stack has grown too large, it calls the C-language routine *__stkover()*.

You will need to modify *__stkover*, since the supplied version simply returns. For example, *__stkover* could print an error message and then exit.

Since compiling with *+m* causes the code to be bigger and execute slower, the final version of your program should be compiled without this option.

2.2 Source level debugging

The *-n* option causes the compiler to generate source level debugging information. This information is passed as special lines and characters in the assembly language output file. The assembler automatically passes source level debugging information through to an object module.

When object modules are linked together into a program, the *-g* option causes the linker to generate a special file that contains the source level debugging information.

These options require the use of an emulator that supports the source level debugging file created by the linker.

When source level debugging information is generated, object modules may increase significantly in size. However, the end program will remain the same size.

2.2.1 Other new features

The compiler now pre-defines the name AZTEC_C, which can be used when writing compiler specific code that it is to distributed.

The compiler now pre-defines the names _LARGE_CODE and _LARGE_DATA when the *+c* and *+d* options are given.

The compiler now supports the enumerated data type.

Structure arguments and return values are now correctly handled.

2.3 New features of the AS68 assembler

The following paragraphs describe new features of the *as68* assembler, v3.4b.

2.3.1 New Processor Support

The assembler is partly redesigned and supports the MC68010, MC68020, and the MC68881 instruction sets and addressing modes in addition to those of the MC68000. By default, the assembler assumes that only the MC68000 instructions are valid. The *MACHINE* and *MC68881* directives enable and/or disable the additional instructions and addressing modes.

2.3.2 Optimization improvements

Space for the squeeze table that's used for optimizing a program is now dynamically allocated, so the -S option is no longer needed or supported.

A new algorithm is used to optimize assembly language code. It is orders of magnitude faster than the old algorithm on large files.

The new optimization algorithm is nonrecursive and therefore no longer requires more than a 4K stack.

All instructions will be considered for squeezing.

2.3.3 Temporary Labels

Temporary labels of the form *n$*, where *n* consists of decimal digits, are now supported. These labels are in effect till the next non-temporary label is encountered. For example:

```
1$:     move.l (a0)+,(a1)+
        dbra   d0,1$
```

2.3.4 Changes To Macros

A number of changes have been made to the implementation of macros. First, the syntax of the macro definition has been expanded to allow the macro name to be an argument of the MACRO directive or to be taken from a label if present. For example, previously a macro could only be defined as:

```
        macro  addnum
```

Now, however it can still be defined this way or as:

```
addnum          macro
```

Macro arguments can now be referenced by either %n or \n. The %0 or \0 argument refers to the extension on the macro directive when invoked. Macro arguments that contain a space or comma can be enclosed in bracketing '<' and '>' characters.

When a backslash is followed by the symbol '@', the assembler generates text of the form ".nnn" where *nnn* has a unique value for each invocation of the macro. This is normally used to generate unique labels within a macro.

The symbol NARG is a special assembler symbol which indicates the number of arguments specified when the macro was invoked. Outside of a macro, the value of NARG is 0.

2.3.5 New operators

The following new operators are supported:

!	- inclusive or
^	- exclusive or
~	- bitwise not
//	- modulo

2.3.6 New directives

The following new directives are supported:

BLANKS

blanks on/off
blanks yes/no
blanks y/n

The *blanks* directive controls whether the assembler will allow blanks or tabs in an instruction's operand field.

The blanks *off* setting treats a blank as the end of the operand field.

The blanks *on* setting allows blanks to be placed between any two complete items. With this setting all comments must be preceded by a ';'.

By default, blanks are *off*, which causes the v3.4b assembler to behave, in this respect, like the v3.30c assembler.

CNOP

label cnop n1,n2

The *cnop* directive forces alignment on any boundary at a particular offset. The first value, *n1*, is an offset while the second value, *n2*, specifies the alignment to be used as the base of the offset. For example, to align to an even word boundary:

```
cnop  0,2
```

while to align to a long word boundary:

```
cnop  0,4
```

and finally to align to a word beyond a long word boundary:

```
cnop  2,4
```

Note that this will only take effect relative to the beginning of the current module's code or data. Normally, the linker will not

align individual modules to long word boundaries. So, for this directive to work, it must either be the first module linked into the program, or else the *+A* option of the linker must be used to force long word alignment of modules.

EQUR

label equr register

This directive allows a register to be referenced by an alternate name. Reference to the new name is made without regard to case.

EVEN

label even

This directive forces alignment to a word (16 bit) boundary.

FAIL

fail

This directive causes the assembler to generate an error for this line. This can be used in macros which detect the incorrect number of arguments and wish to prevent assembly.

FREG

label freg <register list>

This directive is like the REG directive, except that it is used to specify the floating point registers of the MC68881. The list is either composed of the floating point registers *FP0* through *FP7* or of the floating point control registers *FPIAR/FPCR/FPSR*, but not both.

IFC and IFNC

ifc 'string1','string2'
ifnc 'string1','string2'

These conditionals check to see if the two strings are equal. If they are, the *ifc* will assemble the following code, while *ifnc* will skip it.

IFD and IFND

ifd symbol
ifnd symbol

These conditionals check to see if the specified symbol has been defined or not. If the symbol has been defined, then *ifd* will assemble the following code, while *ifnd* will not.

OTHER IFS

ifeq absolute_expression
ifgt absolute_expression
ifge absolute_expression
ifle absolute_expression
iflt absolute_expression
ifne absolute_expression

These conditionals perform a comparison of the value of the absolute expression to zero. If the specified condition is true, then the following assembly language is processed, otherwise it is skipped.

MACHINE

machine MC68000
machine MC68010
machine MC68020

This directive enables or disables the additional instructions and addressing modes associated with different processors in the MC68000 family.

MC68881

mc68881

This directive enables the MC68881 floating point instructions to be recognized and assembled by the assembler.

SECTION

label section name,CODE
label section name,DATA
label section name,BSS

This directive performs the same functions as the *cseg* and *dseg* directives. The name parameter, if present, is ignored at the current time. The type parameter is used to switch from CODE and back again. If only a name parameter is specified, the type defaults to CODE.

SET

label set expression

This directive assigns the value of the absolute expression to the symbol specified by *label*. This definition is similar to the *EQU* directive, with the exception that this symbol's value can be changed with another *SET* directive.

TTL

ttl title_string

This directive sets the title of the current module being assembled. This directive is implemented for compatibility with other assemblers and has no effect at the current time.

XDEF and XREF

xdef symbol
xref symbol

These directives are used to specify the definition and reference of global symbols. Currently these are both mapped onto the *PUBLIC* directive.

2.4 Changes to the linker

The following paragraphs describe features that are new in v3.4b of the *ln68* linker.

2.4.1 Renamed options

Several linker options are now preceded by a plus character (+) instead of a minus (-). These are:

+*R dd* Use address register *dd* for small model operations. *dd* is a decimal value, and default to 5 (ie, address register A5).

+*C xxxx* Set origin of code section to the hex value *xxxx* (default: 0).

+*D xxxx* Set origin of initialized data section to the hex value *xxxx* (default: immediately after the code section).

+*U xxxx* Set origin of the uninitialized data section to the hex value *xxxx* (default: immediately after the initialized data section).

+*S xxxx* Set the size of the stack area to the hex value *xxxx* (default: 2k).

+*J xxxx* Set the program's initial stack pointer to the hex value *xxxx*. (default: stack area immediately follows uninitialized data section, with size specified by +*S* option; stack pointer points to the top of this area).

2.4.2 New options

+*A* Toggle 'long align' mode. When this mode is enabled, each module's code begins on a longword boundary; i.e. on a byte whose address is a multiple of 4. By default, this mode is disabled.

+*Q* Be quiet; i.e. don't list, on the console, each module that is included in a program. By default, the linker issues this list.

2.4.3 Source level debugging

Two new options *-g*, and *-q* have been added to the linker to turn on/off the collection of the symbol table information that will be used by emulators' source level debuggers.

-g Collect source level debug information. This information is put into a file whose name consists of *filename.dbg*, where *filename* is the name specified by the *-o* option or defaults to the name of the first object file listed. The *.dbg* file is automatically looked for when you invoke *sdb*.

-q Turn off the collection of source level debug information for all files following it.

Both options apply only to those files listed after the option on the command line. Both options may be used on the same command line, *-g* will turn on the collection of information for all files after it until the end or a *-q* is encountered. A *-q* will turn off collection of debug information until a *-g* is encountered.

2.5 Library changes

2.5.1 Math libraries

There are now two math libraries supported by Aztec C68k/ROM:

* *m881.lib*, for programs that perform floating point operations using the 68881 coprocessor.
* *m68.lib*, for progfams that perform floating point operations using software emulation.

Source for *m881.lib* is in *m881.arc*. It should be dearchived into a new subdirectory of the *lib* directory, named *m881*.

Source for *m68.lib*, which used to be in one archive named *flt68.arc*, is now in two source archives:

* *mx_ieee.arc*, which should be dearchived into the subdirectory *mx_ieee* of the *lib* directory;
* *math.arc* which should be dearchived into the subdirectory *math* of the *lib* directory.

Changes have also been made to the *libmak68.arc* archive.

2.5.2 The ___stkchk function

The new *___stkchk* function performs stack-depth checking. It is called automatically on entry to functions that have been compiled with *c68*'s new *+m* option. Source for *___stkchk* is in *stkchk.c*, in *rom68.arc*. Before using it, you must customize it.

2.5.3 Changes to *rom68.a68*

Slight changes have been made to the startup routine in *rom68.a68* to support *___stkchk*.

2.5.4 The write() function

The previous version of the *write* function made calls to the CP/M-68k *bdos*. This code has been removed from *write*, thus making it purely a skeleton function. Like the other unbuffered i/o functions, you must flesh out the *write* function in order to use it.

3. Known bugs

* *as68* reports errors on the following:

```
movep.l   d0,0(a0)
```

  It also gets errors on related forms of this instruction, such as movep.w, etc. However, when the displacement is non-zero, no error occurs.

* *as68* doesn't support the *.b* and *.w* extensions for *bsr*, *bra*,, etc. It also doesn't support the *.s* and *.l* extensions for the *jmp* and *jsr* instructions.

* *as68* reports errors when the following is assembled with *-c* and *-d* options:

```
            dseg
regsave     ds.l      16
            cseg
            public    .begin
            entry     .begin
            movem.l   d0-d7/a0-a7,regsave
            dbra      d0,.2
```

  If the code is assembled with *-c*, *-d*, and *-n*, no errors are reported, but the object code doesn't list *.begin* as an entry point.

* When invoked with the *+f8* option, *c68* generates incorrect 68881 code for calls to the floating point functions *sqrt*, *sin*, etc.

* The following code, when compiled with *+f8*, sends *c68* v3.4b into an infinite loop, or makes it crash the OS:

```
long a; double b;
main()
{
        a = b++;
}
```

4. Packaging

Aztec C68k/ROM contains the following files:

c68.exe	Compiler
as68.exe	Assembler
ln68.exe	Linker
lb68.exe	Object module librarian
ord68.exe	Object module orderer
cnm68.exe	Object module utility
obd68.exe	Object module utility
hex68.exe	Binary-to-Intel-hex-record translator
srec68.exe	Binary-to-Motorola-S-record translator
libmak68.arc	Libgen control files
stdio.arc	Source for STDIO functions
misc.arc	Source for MISC functions
mch68.arc	Source for MCH68 functions
rom68.arc	Source for ROM68 functions
mx_ieee.arc	Source for IEEE float emulation functions
math.arc	Source for Transcendental functions
m881.arc	Source for 68881 functions
ctype.h	
errno.h	
fcntl.h	
macros.h	
setjmp.h	
stat.h	
stdio.h	

Using MANX Technical Support

We have put together a set of guidelines to help you take the most advantage of the technical support service offered by MANX. We ask that you read and follow these guidelines to enable us to continue to give you quality technical support.

Have everything with you.

Try to be organized. When using our phone support, have everything you need with you at the time you call. Our goal is to get you the help you need without keeping you on the phone too long. This can save you a lot of time, and if we can keep the calls as short as possible we can take more calls in the day. This can be to your advantage on days when we are busy and it's hard to get through. Also, *have the following information ready* when you call technical support. We will ask you for this information first.

* *Your name.* This is necessary in case we need to get back to you with additional information.
* *Phone number.* In case we have additional information we will be able to contact you. This will never be given to anyone, so you need not worry.
* The *product* you are using, and the *serial number.* If you have a cross compiler please tell us both host and target, even if the problem is with just one side of the system.
* The *revision of the product* you are using. This should include a letter after the number: i.e. 3.20d or 1.06d. THIS IS VERY IMPORTANT. The full version number may be found on your distribution disks or when you run the COMPILER.
* The *operating system* you are using, and also the version.
* The *type of machine* you are using.
* Anything interesting about your machine configuration. ie. ram disk, hard disk, disk cache software etc.

Know what questions you wish to ask.

If you call with a usage question please try to have your questions narrowed down as much as possible. It is easier and quicker for all to answer a specific question than general ones.

Isolate the code that caused the problem.

If you think you have found a bug in our software, try and create a small program that reproduces the problem. If this program is small enough we will take it over the phone, otherwise we would prefer that you mail it to us, using the supplied problem report, or leave it on one of our bbs systems. Once we receive a "bug report" we will attempt to reproduce the problem and if successful we will try to have it fixed in the next release. If we can not reproduce the problem we will contact you for more information.

Use your C language book and technical manuals first.

We have no qualms about helping you with your general C programming questions, but please check with a C language programming book first. This may answer your question quicker and more thoroughly. Also, if you have questions about machine specific code, i.e. interrupts or dos calls, check with that machine's technical reference manual and/or operating system manual.

When to expect an answer.

A normal turn around time for a question is anywhere from 2 minutes to 2 days, depending on the nature of the question. A few questions like tracing compiler bugs may take a little longer. If you can call us back the next day, or when the person you talk to in technical support recommends, we will have an in-depth answer for you. But normally we can answer your questions immediately.

Utilize our mail-in service.

It is always easier for us to answer your question if you mail us a letter (We have included copies of our problem report form for your use). This is especially true if you've found a bug with our compiler or other software in our package. If you do mail your question in, try to include all of the above information, and/or a disk with the problem. Again, please write small test programs to reproduce possible bugs. The address for mail-in reports is P.O. Box 55, Shrewsbury, N.J. 07701. If you have questions/problems concerning C Prime or Apprentice C, mail them to P.O. Box 8, Shrewsbury, N.J. 07701.

Updates, Availability, Prices.

If you have any questions about updates, availability of software, or prices, please call our order desk. They can help you better and faster. You can reach them at...

Outside N.J. --> 1-800-221-0440
Inside N.J. --> 1-201-542-2121 (also for outside the U.S.A.)

Bulletin board system.

For users of Aztec C we have a bulletin board system available. The number is ...

1-(201)-542-2793 This is at 300/1200 bps. (all products)

Answer the questions that will be asked after you are connected. When this is done you will be on the system with limited access. To gain a higher access level send mail to SYSOP. Include in this information your serial number and what product you have. Within approximately 24 hours you should have a higher access level, provided the serial number is valid. This will allow you to look at the various information files and upload/download files.

To use the bulletin board best, please do not put large (> 8 lines) source files onto the news system, which we use for an open forum question/answer area. Instead, upload the files to the appropriate area, and post a news item explaining the problem you are having. Also, the smaller the test program, the quicker and easier it is for us to look into the problem, not to mention the savings of phone time.

When you do post a news item, please date it and sign it. This will be very helpful in keeping track of questions. Try to do the same with uploaded source files.

Phone support, number and hours.

Technical support for Aztec C is available between 10-12 am and 2-6 pm eastern standard time at 1-(201)-542-1795. Phone support is available to registered users of Aztec C with the exception of the Apprentice C and C Prime products. For those products, please use the mail-in support service and send questions/problems to P.O. Box 8, Shrewsbury, N.J. 07701.

These guidelines will aid us in helping you quickly through any roadblocks you may find in your development. Thanks for your cooperation.

Aztec C68k/ROM
Cross Development System

version 3.4
November 1987

Distributed by:
Manx Software Systems, Inc.
P.O. Box 55
Shrewsbury, N.J. 07701
201-542-2121

USE RESTRICTIONS

The components of the Aztec C68k/ROM software development system are licensed software products. Manx Software Systems reserves all distribution rights to these products. Use of these products is prohibited without a valid license agreement. The license agreement is provided with each package. Before using any of these products the license agreement must be signed and mailed to:

Manx Software Systems
P. O. Box 55
Shrewsbury, N. J 07701

The license agreement limits use of these products to one machine. Any uses of these products that might lead to the creation of or distribution of unauthorized copies of these products will be a breach of the licensing agreement and Manx Software Systems will excercise its right to reclaim the original and any and all copies derived in whole or in part from first or later generations and to pursue any appropriate legal actions.

Software that is developed with Aztec C68k/ROM software development system can be run on machines that are not licensed for these products as long as no part of the Aztec C software, libraries, supporting files, or documentation is distributed with or required by the software. In the latter case a licensed copy of the appropriate Aztec C software is required for each machine utilizing the software. There is no licensing required for executable modules that include runtime library routines.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013. DAC #84-1, 1 March 1984. DOD Far Supplement.

COPYRIGHT

DISCLAIMER

Manx Software Systems makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Manx Software Systems reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Manx Software Systems to notify any person of such revision or changes.

TRADEMARKS

Aztec C68k, Manx AS, Manx LN, and Z are trademarks of Manx Software Systems. Amiga is a trademark of Commodore-Amiga, Inc. CP/M-86 is a tradmark of Digital Research. MSDOS is a trademark of Microsoft. PCDOS is a trademark of IBM. UNIX is a trademark of Bell Laboratories. Macintosh is a trademark of Apple Computer.

Manual Revision History

January 1987 ..... First Edition
November 1987 ..... Second Edition

Summary of Contents

68k/ROM-specific Chapters

Host-specific Chapters

(for a list of these chapters, see your release document)

System Independent Chapters

Contents

OVERVIEW

Overview

Aztec C68k/ROM is a set of programs for developing programs in the C programming language; the resulting programs run on ROM- and/or RAM-based systems that use a Motorola 68000-family microprocessor. The development can be done on any of several host systems.

Some of the features of Aztec C68k/ROM are:

* The full C language, as defined in the book *The C Programming Language*, by Brian Kernighan and Dennis Ritchie, is supported.
* An extensive set of user-callable functions is provided, in source form. To use these functions, you must first compile and assemble them, and create libraries of the resulting object modules. To use the standard and/or unbuffered i/o functions, you must write the unbuffered i/o functions.
* Modular programming is supported, allowing the components of a program to be compiled separately, and then linked together.
* Assembly language code can either be combined in-line with C source code, or placed in separate modules which are then linked with C modules.
* Utility programs are provided that generate Motorola S-records and Intel hex records for a program. ROM chips generated from these records will contain the program's code and a copy of its initialized data.
* A ROM program can contain both initialized and uninitialized global and static variables. When the program starts, its initialized variables in RAM will be automatically set from the copy in ROM, and its uninitialized variables will be cleared.

The functions provided with this package are UNIX compatible and are compatible with Aztec C packages provided for other systems. Thus, once you have customized the functions, you can create programs that will run on UNIX-based systems or on other systems supported by Aztec C with little or no change.

Host systems

The Aztec C68k/ROM software runs on several host systems, including:

* PCDOS/MSDOS systems, such as the IBM PC;
* Apple Macintosh;
* Digital Equipment VAX systems that use the Ultrix operating system;

Components

Aztec C68k/ROM contains the following components:

* *c68*, the compiler;
* *as68*, the assembler;
* *ln68*, the linker;
* *lb68*, the object module librarian;
* Source for the library functions;
* Several utility programs.

Preview

This manual is divided into three separate sections, each of which is in turn divided into chapters. The first section presents 68k/ROM-specific information. The second describes host-specific features. The third describes features that are common to all Aztec C packages.

The 68k/ROM-specific chapters and their identifying codes are:

tut describes how to get started with Aztec C68k/ROM: it discusses the installation of Aztec C68k/ROM and gives an overview of the process for turning a C source program into Motorola S-records and Intel hex records;

cc, *as*, and *ln* present detailed information on the compiler, assembler, and linker;

util68k describes the 68k/ROM-specific utility programs that are provided with Aztec C68k/ROM;

libgen describes the creation of object module libraries from the provided source;

tech discusses miscellaneous topics, including C-callable assembly language functions, and C language interrupt handlers.

The contents of the manual's host-specific section varies from host to host. It usually contains a chapter that describes the special utility programs that are provided with your system; this chapter's code has the form *utilxx*, where *xx* identifies the host; for a PCDOS/MSDOS host, for example, the code is *utilpc*.

The System-independent chapters and their codes are:

libov presents an overview of the functions provided with Aztec C;

lib describes the system-independent functions provided with Aztec C68k/ROM;

style discusses several topics related to the development of C programs;

err lists and describes the error messages that are generated by the compiler.

TUTORIAL INTRODUCTION

Chapter Contents

Tutorial Introduction

This chapter describes how to quickly start using your Aztec C68k/ROM cross development software. It discusses the following topics: (1) installing the Aztec C68k/ROM software on your disks; (2) creating object module libraries from the provided source; (3) translating a C program into Motorola S-records or Intel hex code; (4) special features of Aztec C68k/ROM; (5) introduction to the rest of the manual.

Ideally, this chapter should consist of a cookbook set of steps that you can follow to get started using Aztec C68k/ROM. However, since one of those steps is a long and involved one, (ie, to modify the library functions and then generate libraries), we recommend that you follow the first step, which leads you through the installation of Aztec C68k/ROM on your system, and then simply read the rest of chapter to get a idea of how programs are developed using Aztec C68k/ROM. Then you can read the Library Generation chapter, make any needed revisions to the library function source, and generate your libraries. Finally, you can translate a C program into a ROM-burnable format, by following the steps in this chapter.

1. Installing Aztec C68k/ROM

To install Aztec C68k/ROM on your system, copy the files from the distribution media (disk or tape) onto your disks.

If your system is one (such as the IBM PC running PCDOS, or a UNIX system) that supports a hierarchical directory structure, we recommend that you place the Aztec C68k/ROM software in a set of related directories, as shown in the following diagram.

Directory		*Contents*
C68		
	BIN	executable programs
	INCLUDE	header files
	LIB	object module libraries
	STDIO	stdio.arc files
	MISC	misc.arc files
	MCH68	mch68.arc files
	ROM68	rom68.arc files
	MATH	math.arc files
	MX_IEEE	mx_ieee.arc files
	M881	m881.arc files

Copy the Aztec C68k/ROM files into the directories as follows:

* Into the BIN directory, copy all executable Aztec C68k/ROM programs.

* Into the INCLUDE directory, copy all "include files" (that is, files having extension *.h*).

* Into the LIB directory, copy the source archive *libmake.arc*. The libraries that you create will reside in this directory.

  Extract the files from this archive using the *arcv* command, and then delete *libmake.arc* from the LIB directory.

  To extract files from *libmake.arc* follow these steps: (1) make sure that the BIN directory is in the path of directories that will be searched by the operating system for programs (on PCDOS and UNIX, this means adding the BIN directory name to the PATH environment variable); (2) enter the appropriate command to make LIB the default or current directory (for example, on PCDOS this command is *cd \C68\ LIB*); (3) enter the command *arcv libmake.arc*.

* Into the STDIO, MCH68, ..., and ROM68 directories, copy the corresponding source archive (for example, copy *stdio.arc* into the STDIO directory, *mch68.arc* into MCH68, and so on).

  Extract the files from each archive using *arcv*, and then delete the archive.

  Each of these directories contains the source and object modules generated from the corresponding source archive file. For example, the source files in STDIO were extracted from the *stdio.arc* source archive file by the *arcv* program.

2. Creating Object Module Libraries

The functions that are provided with Aztec C68k/ROM are in source form. Before you can create an executable program using C68k/ROM, you must compile and assemble the functions and generate object module libraries that contain them, after first making any needed modifications. For more information, see the Library Generation chapter.

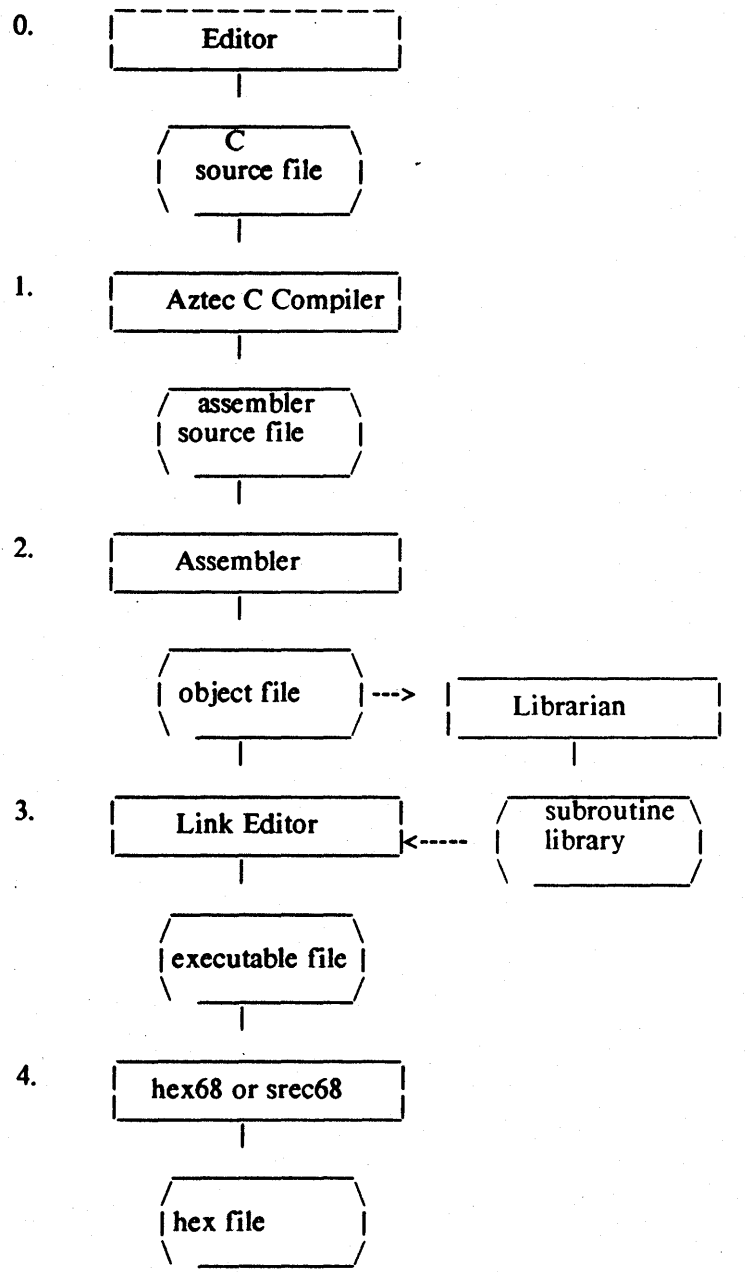

Figure 1: Program Development with Aztec C68k/ROM

3. Translating a program into hex code

In this section we will lead you through the steps necessary to translate a sample C program named *exmpl.c* into hex code that can be burned into ROM. For a diagram of this procedure, see figure 1.

The code for this program will reside in ROM, beginning at memory location 0. Its data will reside in RAM, beginning at location 0x8000.

3.1 Step 0: Create the Source Program

The first step to creating a C program is, of course, to create a disk file containing its source. For this, you can use any text editor. We'll assume the source exists, in the file *exmpl.c*.

3.2 Steps 1 and 2: Compile and Assemble

To compile and assemble *exmpl.c* enter the following command:

```
c68 exmpl.c
```

This first starts the *c68* compiler, which translates the C source that's in *exmpl.c* into assembly language source. When done, *c68* starts the *as68* assembler. *as68* assembles the assembly language source for the sample program, translating it into object code and writing the object code to the file *exmpl.r* in the current directory. When done, *as68* deletes the file that contains the assembly language source, since it is no longer needed.

There are several compiler options that define a module's characteristics. For this example, we have let these options assume their default values. Later in this chapter we introduce some of these characteristics.

3.3 Step 3: Link

The object code version of the *exmpl* program must next be linked to needed functions that are in the *c68.lib* library of object modules and converted into a loadable format.

Before entering this command, you must set the CLIB68 environment variable, to define the directory that contains the object module libraries. For example, on PCDOS, if the libraries are in *e:\c68\lib*, the command to define CLIB68 is

```
set CLIB68=e:\c68\lib\
```

Note the terminating slash: this is usually required, because of the way the linker builds the complete name of a library that is partially identified using the linker's *-l* option. This is described below.

The command to link the sample program is:

```
ln68 +d 8000 -o exmpl rom68.r exmpl.r -lc68
```

There's several parameters to this command, so let's go through them, one at a time.

3.3.1 Positioning code, data, and stack: the +D, +U, +C, +S, & +J options

The linker organizes a program into three sections:

Code — Contains the program's executable code;

Initialized data — Contains those of the program's global and static variables that are assigned an initial value (e.g. *static int var=1*);

Uninitialized data. — Contains the program's other global and static variables.

The linker supports options that allow you to position these segments in memory. The *+D 8000* option used in the above command sets the starting address of the program's initialized data to 0x8000.

The linker's *+U* option sets the starting address of the program's uninitialized data. This option wasn't used in the above command, so the uninitialized data begins at its default address; i.e. immediately above the initialized data.

The linker's *+C* option sets the starting address of the program's code segment. This option wasn't used in the above command, so the code area begins at its default starting address; i.e. location 0.

The linker's *+S* and *+J* options set the starting address of the program's stack pointer and the size of the stack area. These options weren't used in the above command, so they assume their default values: the stack area begins immediately after the uninitialized data area, the area is 2k bytes long, and the stack pointer initially points at the top of this area.

3.3.2 Naming the output file: the -O option

The *-O exmpl* option tells the linker to place the linked program in the file named *exmpl*. If this option wasn't used, the linker would have derived the name of the output file from that of the first object module, by deleting its extension.

3.3.3 The input object module files

rom68.r and *exmpl.r* are the names of two files whose object modules are to be included in the program.

rom68.r contains the startup routine for both the program and for the system, and contains statements that pre-initialize the system's startup and interrupt vectors. It's listed first so that its code, and hence these vectors, will be loaded at the beginning of the program's code

segment; since the code segment begins at location 0, these vectors will then be correctly positioned in memory.

When linking programs with *rom68.r* as the first listed module, you'll usually have to explicitly specify the name of the output file, using the -O option. If you don't, the linker will place all such programs in the same file; i.e. in the file named *rom68.*

3.3.4 Libraries and the -L option

The *-Lc68* option tells the linker to search the *c68.lib* library that's in the directory defined by the CLIB68 environment variable for needed functions.

As you can see, the *-L* option doesn't completely define the name of a library file; the linker generates the complete name by taking the letters that follow the *-L*, prepending them with the value of the CLIB68 environment variable, and appending the letters *.lib*. Thus, when CLIB68 has the value *e:\c68\lib*, the *-Lc68* option specifies the library whose complete file name is *e:\c68\lib\c68.lib.*

During the link step, the linker will search the libraries specified to it for modules containing needed functions; when such a module is found, the linker will include the module in the executable file it's building.

All C programs need to be linked with *c68.lib* (or an equivalent, as described below). This library contains the non-floating point functions that are defined in the System Independent Functions chapter. It also contains "internal" functions that are called by compiler-generated code.

If a program performs floating point operations, it must also be linked with the *m68.lib* math library (or an equivalent, as described below).

When a program is linked with a math library, that library must be specified before *c68.lib*. For example, if *exmpl.c* performed floating point, the following would link it:

```
ln68 +d 8000 -o exmpl rom68.r exmpl.r -lm68 -lc68
```

3.4 Step 4: Convert to Motorola S-records or Intel hex records

The next step is to convert the memory image generated by the linker into Motorola S-records or Intel hex records, using *srec68* or *hex68*, respectively. In the following discussion, we'll generate S-records using *srec68*. At the end of the section, we show how to generate Intel hex records using *hex68.*

To generate Motorola S-records for the program, enter the following command:

srec68 exmpl

When the records generated by this command are fed into a ROM programmer, the resulting ROM code will contain the program's code segment followed by a copy of its initialized data segment.

Note: when the system is started, its RAM contains random values; the Aztec startup routine sets up the RAM-resident initialized data segment from the ROM-resident copy.

These commands generate one or more files, each of which contains S-records for one 2k-byte, successively-higher addressed section of the program's code and initialized data. The files are *exmpl.m00* (containing first ROM chip's S-records), *exmpl.m01* (containing the second ROM chip's S-records), and so on.

srec68 has several additional features. For example, you can explicitly define the size of each ROM chip, using the -P option; and you can have it place a program's even-addressed and odd-addressed bytes in separate ROM chips, using the -E and -O options.

hex68 behaves just like *srec68*, except that it generates Intel hex records instead of Motorola S-records, and the extensions of the generated files are slightly different. For example, the command to convert *exmpl* into Intel hex records is:

hex68 exmpl

This generates the files *exmpl.h00*, *exmpl.h01*, and so on; where each file contains hex records for successively-higher-addressed 2k-byte ROM chips.

For complete descriptions of *srec68* and *hex68*, see the Utility Programs chapter.

4. Special features of Aztec C68k/ROM

That concludes our step-by-step, cookbook introduction to Aztec C68k/ROM. In the following paragraphs, we want to introduce several special features of Aztec C68k/ROM.

4.1 Memory models

Aztec C68k/ROM allows you to define, when you compile and assemble a module, the "memory model" that the module will use. A module's memory model affects the module's speed, size, and the amount of data it can access. By default, a module will use the *small code, small data* memory model, which makes it small and fast, but you can override this using compiler and assembler options.

Library modules have memory models, too. The makefiles that are provided with Aztec C68k/ROM make two versions each of the "c68" and "m68" libraries: in one version of a library, the modules all use the *small code, small data* memory model; in the other, they use *large code*,

large data. If desired, you can modify these makefiles to make other versions of these libraries, whose modules use different combinations of memory models.

Here's where to go for more information:

* For a complete description of memory models, see the Compiler chapter.
* The compiler options for selecting a module's memory model are +C and +D; they are discussed in the Options section of the Compiler chapter;
* The assembler options for selecting the default memory model are -C and -D; they are discussed in the Options section of the Assembler chapter;
* The assembler directives *near* and *far* also define memory models; they are discussed in the Programmer section of the Assembler chaper;
* The creation of libraries is discussed in the Library Generation chapter.

4.2 Register usage

By default, a program's register usage is as follows:

* Temporary results: data registers D0-D3; address registers A0-A2.
* Register variables: data registers D4-D7; address registers A3 and A4.
* Small model support register: A5.
* Frame pointer: A6.
* Stack pointer: A7.

Using the compiler's +R option, you can define the registers used for temporary results, register variables, and the frame pointer.

Using the linker's +R option, you can define the register used to support modules that use a small memory model.

The makefiles that are provided with Aztec C68k/ROM generate libraries whose modules use the default registers.

5. Where to go from here

In this chapter, we've just begun to describe the features of Aztec C68k/ROM.

One chapter that you must read is the Library Generation chapter, which discusses the generation of object module libraries from the source that comes with Aztec C68k/ROM.

We encourage you to use the *make* program-maintenance program to generate libraries, if such a program is available for your host system. To provide this encouragement, Aztec C68k/ROM provides "makefiles" that can be used by UNIX-compatible *make* programs. If your host system is one, such as PCDOS, that doesn't have its own *make* program, and if the Aztec *make* is available for your system, it will be included in your Aztec C68k/ROM package.

For more information on the sections of a program, see the Linker chapter.

The *srec68* and *hex68* programs support several options that haven't been discussed in this introduction. For a complete description of these programs, see the Utility Programs chapter.

The Technical Information chapter contains miscellaneous information on several topics, including the writing of assembly language functions and interrupt handlers.

You should also read the Compiler, Assembler, and Linker chapters, to become familiar with all the options that these programs provide.

THE COMPILER

Chapter Contents

The Compiler

This chapter describes *c68*, the Aztec C compiler for generic Motorola 68000-based systems. It is not intented to be a complete guide to the C language; for that, you must consult other texts. One such text is *The C Programming Language*, by Kernighan and Ritchie. The compilers were implemented according to the language description in the Kernighan and Ritchie book.

This description of the compilers is divided into four subsections, which describe how to use the compiler, compiler options, information related to the writing of programs, and error processing.

1. Compiler Operating Instructions

c68 is invoked by a command of the form:

```
c68 [-options] filename.c
```

where [-options] specify optional parameters, and *filename.c* is the name of the file containing the C source program. Options can appear either before or after the name of the C source file.

The compiler reads C source statements from the input file, translates them to assembly language source, and writes the result to another file.

1.1 The C source file

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

```
c68 text
```

1.2 The output files

1.2.1 Creating an object code file

Normally, when you compile a C program you are interested in the relocatable object code for the program, and not in its assembly language source. Because of this, the compiler by default writes the assembly language source for a C program to an intermediate file and then automatically starts the assembler. The assembler then translates the assembly language source to relocatable object code, writes this code to a file, and erases the intermediate file.

By default, the object code generated by a *c68*-started assembly is sent to a file whose name is derived from that of the file containing the C source by changing its extension to *.r*. This file is placed in the

directory that contains the C source file. For example, if the compiler is started with the command

```
c68 prog.c
```

the file *prog.r* will be created, containing the relocatable object code for the program.

The name of the file containing the object code created by a compiler-started assembler can also be explicitly specified when the compiler is started, using the compiler's -O option. For example, the command

```
c68 -O myobj.rel prog.c
```

compiles and assembles the C source that's in the file *prog.c*, writing the object code to the file *myobj.rel*.

When the compiler is going to automatically start the assembler, it by default writes the assembly language source to a temporary file named *ctmpxxx.xxx*, where the x's are replaced by digits in such a way that the name becomes unique. This temporary file is placed in the directory specified by the environment variable *CCTEMP*. If this variable doesn't exist, the file is placed in the current directory.

When *CCTEMP* exists, the complete name of the temporary file is generated by simply prefixing its value to the ctmpxxx.xxx name. For example, if *CCTEMP* has the value

```
/RAM/TEMP/
```

then temporary files are placed in the /RAM/TEMP/ directory.

For a description on the setting of environment variables, see your operating system manual.

If you are interested in the assembly language source, but still want the compiler to start the assembler, specify the option -T when you start the compiler. This will cause the compiler to send the assembly language source to a file whose name is derived from that of the file containing the C source by changing its extension to *.a*. The C source statements will be included as comments in the assembly language source. For example, the command

```
c68 -T prog.c
```

compiles and assembles *prog.c*, creating the files *prog.a* and *prog.r*.

1.2.2 Creating just an assembly language file

There are some programs for which you don't want the compiler to automatically start the assembler. For example, you may want to modify the assembly language generated by the compiler for a particular program. In such cases, you can use the compiler's *-A* option to prevent the compiler from starting the assembler.

When you compile a program using the *-A* option, you can tell the compiler the name and location of the file to which it should write the assembly language source, using the *-O* option.

If you don't use the *-O* option but do use the *-A* option, the compiler will send the assembly language source to a file whose name is derived from that of the C source file by changing the extension to *.a* and place this file in the same directory as the one that contains the C source file. For example, the command

```
c68 -A prog.c
```

compiles, without assembling, the C source that's in *prog.c*, sending the assembly language source to *prog.a*.

As another example, the command

```
c68 -A -O temp.asm prog.c
```

compiles, without assembling, the C source that's in *prog.c*, sending the assembly language source to the file *temp.asm*.

When the -A option is used, the option -T causes the compiler to include the C source statements as comments in the assembly language source.

1.3 *#include* Files

1.3.1 Searching for *#include* files

You can make the compiler search for *#include* files in a sequence of directories, thus allowing source files and *#include* files to be contained in different directories.

Directories can be specified with the -I compiler option, and with the INCL68 environment variable. The compiler itself also selects a few areas to search. The maximum number of searched areas is eight.

If the file name in the #include statement specifies a directory, just that directory is searched.

1.3.1.1 The -I option.

A -I option defines a single directory to be searched. The area descriptor follows the -I, with no intervening blanks. For example, the following *-I* option tells the compiler to search the */ram/include* directory:

```
-I/ram/include
```

1.3.1.2 The INCL68 environment variable

The INCL68 environment variable also defines directories to be searched for #include files. The string associated with this variable consists of the names of the directories to be searched, with each pair separated by a semicolon. For example, on PCDOS the following

command sets INCL68 so that the compiler will search for include files in directories \C68\INCLUDE and \DVR\INCLUDE:

```
set INCL68=\C68\INCLUDE;\DVR\INCLUDE
```

For a description of the command that's used on your system to set environment variables, see your operating system manual.

1.3.1.3 The search order for include files

Directories are searched in the following order:

1. If the #include statement 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 area isn't automatically searched.
2. The directories defined in -I options are searched, in the order listed on the command line.
3. The directories defined in the INCL68 environment variable are searched, in the order listed.

1.3.2 Precompiled #*include* Files

To shorten compilation time, the compiler supports precompiled #*include* files.

To use this feature, you first compile frequently-used header files, specifying the *+h* option; this causes the compiler to write its symbol table, which contains information about the contents of the header files, to a disk file. Then, when you compile a module that #*include*s some of these header files, you specify the *+i* option; this causes the compiler to load into its symbol table the pre-compiled symbol table information about the header files. When the compiler encounters a #*include* statement of a header file for which it has already loaded pre-compiled symbol table information, it ignores the #*include* statement. This ignoring occurs even if the #*include* file was nested within another #*include* file in the C source from which the pre-compiled symbol table was generated.

The compiler does much less work when it loads pre-compiled information into its symbol table than when it generates the same information from C source, and hence using pre-compiled #*include* files can considerably shorten the time required to compile a module.

The *+H* option tells the compiler to write its symbol table to a file. The name of the file immediately follows the *+H*, with no intervening spaces. For example, you might create a file named *x.c* that consists just of #*include* statements for all the header files that you want pre-compiled. You could then generate a file named *include.pre* that contains the symbol table information for these header files by entering the following command:

```
c68 +Hinclude.pre x.c
```

The *+I* option tells the compiler to read pre-compiled symbol table information from a file. The name of the file immediately follows the *+I*, with no intervening spaces. For example, to compile the file *prog.c* that accesses the header files that were defined in *x.c*, and to have the compiler preload the symbol table information for these files from *include.pre*, enter the following command:

```
c68 +Iinclude.pre prog.c
```

1.4 Memory Models

The memory model used by a program determines how the program's executable code makes references to code and data. This in turn indirectly determines the amount of code and data that the program can have, the size of the executable code, and the program's execution speed.

Before getting into the details of memory models, we want to describe the sections into which a C68k-generated program is organized. The sections of a program are these:

* *code*, containing the program's executable code;
* *data*, containing its global and static data;
* *stack*, containing its automatic variables, control information, and temporary variables;
* *heap*, an area from which buffers are dynamically allocated.

There are two attributes to a program's memory model: one attribute specifies whether the program uses the *large data* or the *small data* memory model; the other attribute specifies whether the program uses the *large code* or *small code* memory model.

1.4.1 *large data* versus *small data*

The fundamental difference between a *large data* and a *small data* program concerns the way that instructions access data segment data: a *large data* program accesses the data using position-dependent instructions; a *small data* program accesses the data using position-independent instructions. An instruction makes position-dependent reference to data in the data segment by specifying the absolute address of the data; it makes a position-independent reference to data in the data segment by specifying the location as an offset from a reserved address register. Other differences in *large data* and *small data* programs result from this fundamental difference; these other differences are:

* There is no limit to the amount of global and static data that a *large data* program can have. A *small data* program, on the other hand, can have at most 64k bytes of global and static data.

* For a *small data* program, an address register must be reserved to point into the middle of the data segment. For a *large data* program, an instruction that wants to access data in the data segment contains the absolute address of the data, and hence doesn't need this address register.

* A code segment is larger when its program uses *large data* than when it uses *small data*, because a reference to data in a data segment occupies a 32-bit field in a *large data* instruction, and occupies a 16-bit field in a *small data* instruction.

* A program is slower when it uses *large data* than when it uses *small data*, because it takes more time for an instruction to access data when it specifies the absolute address of the data than when it specifies the data's offset offset from an address register.

1.4.2 *large code* versus *small code*

The fundamental difference between a *large code* and a *small code* program concerns the way that instructions in the program refer to locations that are located in the code segment: for a *large code* program the reference is made using position-dependent instructions; for a *small code* program, the reference is made using position-independent instructions. An instruction makes position-dependent reference to a code segment location by specifying the absolute address of the location; it makes a position-independent reference to a code segment location by specifying the location as an offset from the current program counter. Other differences in *large data* and *small data* programs result from this fundamental difference; these other differences are:

* The size of a code segment is unlimited for both *large code* and *small code* programs. An instruction in a *large code* program can directly call or jump to the location, regardless of its location in the code segment.

  An instruction in a *small code* program can only directly call or jump to locations that are within 32k bytes of the instruction. To allow instructions in *small code* programs to transfer control to any location, regardless of its location in the code segment, a "jump table", which is located in the program's data segment, is used. If a location to which an instruction wants to transfer control is more than 32k bytes from the instruction, the transfer is made indirectly, via the jump table: the instruction calls or jumps to an entry in the jump table, which in turn jumps to the desired location. A jump instruction in a jump table entry refers to a code segment location using an absolute, 32-bit address, and hence can directly access any location in the program's code

segment.

When a *small code* program is linked, the linker automatically builds the jump table: if the location to which an instruction wants to transfer control is outside the instruction's range, the linker creates a jump table entry that jumps to the location and transforms the pc-relative instruction into a position-independent call or jump to the jump table entry.

* A code segment can contain data as well as executable code. An instruction in a *large code* program can access data located anywhere in the code segment, because it accesses code segment data using position-dependent instructions, in which the location is referred to using a 32-bit, absolute address. An instruction in a *small code* program can only access code segment data that is located within 32k bytes of the instruction.

* For a *small code* program to access the jump table, an address register needs to be reserved and set up to point into the middle of the program's data segment; if the program also uses *small data*, the same address register is used for both jump table accesses and normal accesses of data segment data. For a *large code* program, this address register is not needed for the referencing of locations in the code segment.

* A code segment is larger when its program uses *large code* than when it uses *small code*, because instructions that reference code segment locations by specifying an absolute address use a 32-bit field to define the location, whereas instructions that reference data by specifying a pc-relative address or an offset from an index register use a 16-bit field to define the location.

* A program is usually slower when it uses *large code* than when it uses *small code*, because it takes more time for an instruction to reference a code segment location when it specifies the absolute address of the data than when it specifies the location in a pc-relative form.

A large *small code* program that has lots of indirect transfers of control via the jump table may not differ much in execution time from a *large code* version of the same program, since the *small code* indirect transfer via the jump table will take more time than the *large code* direct transfer.

1.4.3 Selecting a module's memory model

You define the memory model to be used by a module when you compile the module, by specifying or not specifying the following options:

+C Module uses *large code.* If this option isn't specified the module will use *small code.*

+D Module uses *large data.* If this option isn't specified the module will use *small data.*

For example, the following commands compile *prog.c* to use different memory models:

c68 prog	small code, small data
c68 +C prog	large code, small data
c68 +D prog	small code, large data
c68 +C +D	large code, large data

1.4.4 Libraries

The Aztec C68k functions are provided in source form, with "makefiles" that simplify the task of generating object module libraries. The supplied versions of the makefiles can create small code, small data and large code, large data versions of the libraries *c68.lib* and *m68.lib.*

1.4.5 Multi-module programs

The modules that you link together to form an executable program can use different memory models, with the following caveat.

When large data and small data modules are linked together, the linker will create an arbitrarily large data segment, without attempting to sort the data into those that are accessed by large data modules and those that are accessed by small data modules. When the program is running, an address register that you specify at link time will point into the middle of this data segment. This register is used by the small data modules to access data.

Here's the caveat: data that the small data modules attempt to access must be within 32k bytes of the location pointed at by this address register. The linker will detect data accesses by small data modules for which this condition isn't satisfied, and issue a message. If you get this message, try reordering the order in which the linker encounters them if that doesn't solve the problem, you'll have to recompile the small data modules, making them use large data.

2. Compiler Options

There are two types of options in Aztec C compilers: machine independent and machine dependent. The machine-independent options are provided on all Aztec C compilers. They are identified by a leading minus sign.

The Aztec C compiler for each target system has its own, machine-dependent, options. Such options are identified by a leading plus sign.

The following paragraphs first summarize the compiler options and then describe them in detail.

2.1 Summary of options

2.1.1 Machine-independent Options

-A Don't start the assembler when compilation is done.

-Dsymbol[=value]
Define a symbol to the preprocessor.

-Idir1;dir2;...
Search directories *dir1*, *dir2*, ... for #include files.

-O file Send output to *file*.

-S Don't print warning messages.

-T Include C source statements in the assembly code output as comments. Each source statement appears before the assembly code it generates.

-B Don't pause after every fifth error to ask if the compiler should continue. See the Errors subsection for details.

-Enum Use an expression table having *num* entries.

-Lnum Use a local symbol table having *num* entries.

-Ynum Use a case table having *num* entries.

-Znum Use a literal table having *num* bytes.

2.1.2 Special Options for the 68k processor

+B Don't generate the statement "public .begin"

+C Generate code that uses the "large code" memory model. For information on +C and the related +D option, see the Operator Information section.

+D Generate code that uses the "large data" memory model.

+Hfile Write symbol table to *file*. For information on +H and the related +I option, see the Operator Information

section.

+Ifile Read pre-compiled symbol table from *file*.

+L *int* variables and constants are 32 bits long. If this option isn't used, they are 16 bits long.

+Q Put character string constants in the data segment. If +Q isn't specified, string constants are placed in the code segment.

+RFx Use address register *x* as the frame pointer (default: A6).

+RRxxx For register variables, use registers defined by the decimal number *xxx* (default: D4-D7/A3-A4).

+RSxxx On function entry, always save registers defined by the decimal number *xxx* (default: none).

+RTxxx For temporary results, use registers defined by the decimal number *xxx* (default: D0-D3/A0-A2).

+RUx Set name underscore mode as defined by *x*: negative for preceding underscore (default), zero for none, positive for trailing.

2.2 Detailed description of the options

2.2.1 Machine-independent options

2.2.1.1 The -D Option (Define a macro)

The *-D* option defines a symbol in the same way as the preprocessor directive, *#define*. Its usage is as follows:

```
c68 -Dmacro[=text] prog.c
```

For example,

```
c68 -DMAXLEN=1000 prog.c
```

is equivalent to inserting the following line at the beginning of the program:

```
#define  MAXLEN  1000
```

Since the *-D* option causes a symbol to be defined for the preprocessor, this can be used in conjunction with the preprocessor directive, *#ifdef*, to selectively include code in a compilation. A common example is code such as the following:

```
#ifdef  DEBUG
    printf("value: %d\n", i);
#endif
```

This debugging code would be included in the compiled source by the following command:

```
c68 -dDEBUG program.c
```

When no substitution text is specified, the symbol is defined to have the numerical value 1.

2.2.1.2 The -I Option (Include another source file)

The *-I* option causes the compiler to search in a specified directory for files included in the source code. The name of the directory immediately follows the -I, with no intervening spaces. For more details, see the Compiler Operating Instructions, above.

2.2.1.3 The -S Option (Be Silent)

The compiler considers some errors to be genuine errors and others to be possible errors. For the first type of error, the compiler always generates an error message. For the second, it generates a warning message. The *-S* option causes the compiler to not print warning messages.

2.2.1.4 The Local Symbol Table and the -L Option

When the compiler begins processing a compound statement, such as the body of a function or the body of a *for* loop, it makes entries about the statement's local symbols in the local symbol table, and

removes the entries when it finishes processing the statement. If the table overflows, the compiler will display a message and stop.

By default, the local symbol table contains 40 entries. Each entry is 26 bytes long; thus by default the table contains 520 bytes.

You can explicitly define the number of entries in the local symbol table using the *-L* option. The number of entries immediately follows the *-L*, with no intervening spaces. For example, the following compilation will use a table of 75 entries, or almost 2000 bytes:

```
c68 -L75 program.c
```

2.2.1.5 The Expression Table and the -E Option

The compiler uses the expression table to process an expression. When the compiler completes its processing of an expression, it frees all space in this table, thus making the entire table available for the processing of the next expression. If the expression table overflows, the compiler will generate error number 36, "no more expression space", and halt.

By default, the expression table contains 80 entries. Each entry is 14 bytes long; thus by default the table contains 1120 bytes.

You can explicitly define the number of entries in the expression table using the *-E* option. The number of entries immediately follows the *-E*, with no intervening spaces. For example, the following compilation will use a table of 20 entries:

```
c68 -E20 program.c
```

2.2.1.6 The Case Table and the -Y Option

The compiler uses the case table to process a switch statement, making entries in the table for the statement's cases. When it completes its processing of a switch statement, it frees up the entries for that switch. If this table overflows, the compiler will display error 76 and halt.

For example, the following will use a maximum of four entries in the case table:

```
switch (a) {
case 0:                    /* one */
   a += 1;
   break;
case 1:                    /* two */
   switch (x) {
   case 'a':               /* three */
      func1 (a);
      break;
   case 'b':               /* four */
      func2 (b);
      break;
   }                  /* release the last two */
   a = 5;
case 3:                  /* total ends at three */
   func2 (a);
   break;
}
```

By default, the table contains 100 entries. Each entry is four bytes long; thus by default, the table occupies 400 bytes.

You can explicitly define the number of entries in the case table using the compiler's *-Y* option. The number of entries immediately follows the *-Y*, with no intervening spaces. For example, the following compilation uses a case table having 50 entries:

```
c68 -Y50 file
```

2.2.1.7 The String Table and the -Z Option

When the compiler encounters a "literal" (that is, a character string), it places the string in the literal table. If this table overflows, the compiler will display error 2, "string space exhausted", and halt.

By default, the literal table contains 2000 bytes.

You can explicitly define the number of bytes in this table using the compiler's *-Z* option. The number of bytes immediately follows the *-Z*, with no intervening spaces. For example, the following command will reserve 3000 bytes for the string table:

```
c68 -Z3000 file
```

2.2.1.8 The Macro/Global Symbol Table

The compiler stores information about a program's macros and global symbols in the Macro/Global Symbol Table. This table is located in memory above all the other tables used by the compiler. Its size is set after all the other tables have been set, and hence can't be set by you. If this table overflows, the compiler will display the message "Out of Memory!" and halt. You must recompile, using smaller sizes for the other tables.

2.2.2 Special Options for the 68k processor

2.2.2.1 The +B Option

Normally when compiling modules, the compiler generates a reference to the entry point named *.begin*. Then when modules are linked into a program, the reference causes the linker to include in the program the library module that contains *.begin*.

The *+B* option prevents the compiler from generating this reference.

For example, if you want to provide your own entry point for a program, and its name isn't *.begin*, you should compile the program's modules with the *+B* option. If you don't, then the program will be bigger than necessary, since it will contain your entry point module and the standard entry point module. In addition, the linker by default sets at the program's base address a jump instruction to the program's entry point; if it finds entry points in several modules, it will set the jump to the last one encountered.

2.2.2.2 The +C and +D options

These options are discussed in the first section of this chapter.

2.2.2.3 The +L option

The +L option causes a program's *int* variables and constants to be 32 bits long, instead of the 16 bit default length. This option has no effect on the length of a module's other integer variables: variables of type *short* and *long* are always 16 and 32 bits long, respectively.

We recommend that you use the +L option sparingly, if at all, because it makes a program larger and slower.

2.2.2.4 The +RF option - Define the frame pointer register

During execution of a function, a "frame" of information about the function is on the stack. An address register points to the "frame" of the currently-active function, and is used by compiler-generated code to access information in this function's frame.

You can define, using the compiler's +RF option, the address register that will contain the frame pointer. The decimal number of this address register immediately follows the "+RF", with no intervening spaces. For example, the following option tells the compiler to use address register A5 as the frame pointer:

```
+rf5
```

If this option isn't specified, address register A6 is used as the frame pointer.

2.2.2.5 The +RR option - Define register variables' registers

The "+RR" option defines the registers that can be used for a C function's register variables. These registers are specified by the decimal number that immediately follows the "+RR". Each register has a number associated with it, and the number that follows the "+RR" is the sum of all the selected registers' numbers. The registers that can be used to hold register variables, and their associated numbers, are:

Register	Number
D2	1
D3	2
D4	4
D5	8
D6	16
D7	32
A2	64
A3	128
A4	256
A5	512
A6	1024

For example, to define registers D4-D7/A3-A5 as register variables, you would add the numbers for these registers:

4+8+16+32+128+256+512=956,

and then specify, when compiling, the option

+rr956

If you don't specify the +RR option, the compiler uses a default value of 444. This is the sum 4+8+16+32+128+256, which allows registers D4-D7/A3-A4 to be used for register variables.

2.2.2.6 The +RS option - Specify registers to be saved

On entry to a function, the contents of the registers that hold the function's register variables are pushed on the stack. Normally, just those registers that contain the function's register variables are saved; for example, if D4-D7/A3-A4 are available for use as register variables but the function only declares one register variable, then just one register is saved on entry to the function.

The +RS option tells the compiler to generate code that will automatically save specified registers, whether or not they are used for the function's register variables. These registers are specified in a decimal number that immediately follows the +RS. The number has the same format as for the +RR option: it's a sum of numbers, each of which defines one register. The numbers for the registers are the same as for the +RR option.

For example, the following option tells the compiler to generate code that will automatically save A5 on entry to a function:

```
+rs512
```

If this option isn't specified, no extra registers will be saved on entry to a function.

2.2.2.7 The +RT option - Define registers that can hold temps

During the execution of compiler-generated code, registers are used to hold temporary values. This option defines those registers, in a decimal number that immediately follows the "+RT". Each possible register that can be used is assigned a number, and the number that follows the +RT is the sum of the numbers for those registers that can be used for temporaries. The registers and their numbers are:

register	number
D0	1
D1	2
D2	4
D3	8
A0	16
A1	32
A2	64
A6	128

For example, if D0-D2/A0 are available for temporaries, the following +RT option would be used:

```
+rt23
```

If this option isn't specified, registers D0-D3/A0-A2 will be used for temporaries.

2.2.2.8 The +RU option - Define underscore mode

When the compiler translates the name of a function or global variable into assembler, it does so by pre-pending an underscore to the C name, post-pending an underscore, or by using the C name as is. The +RU option defines, in a number that immediately follows the +RU, which of these choices the compiler should use: a negative value prepends the underscore, zero causes no underscore to be added, and a positive value postpends the underscore.

For example, the following option causes the compiler to place an underscore before all names:

```
+ru-1
```

If this option isn't used, the compiler will prepend an underscore.

3. Writing programs

The previous sections of this description of the compiler discussed operational features of the compiler; that is, presented information that an operator would use to compile a C program. In this section, we want to present information of interest to those who are actually writing programs.

3.1 Supported Language Features

Aztec C supports the entire C language as defined in *The C Programming Language* by Kernighan and Ritchie. This now includes the bit field data type.

The following paragraphs describe features of the standard C language that are supported by Aztec C but that aren't described in the K & R text.

3.2 Structure assignment

Aztec C supports structure assignment. With this feature, a program can cause one structure to be copied into another using the assignment operator.

For example, if *s1* and *s2* are structures of the same type, you can say:

```
s1 = s2;
```

thus causing the contents of structure s1 to be copied into structure s2.

Unlike other operators, the assignment operator doesn't have a value when it's used to copy a structure. Thus, you can't say things like "a = b = c", or "(a=b).fld" when a, b, and c are structures.

3.3 Structure Passing

Aztec C68k allows a structure to be passed from one function to another function; but a function cannot return a structure as its value.

3.4 Line continuation

If the compiler finds a source line whose last character is a backslash, \, it will consider the following line to be part of the current line, without the backslash. For example, the following statements define a character array containing the string "abcdef":

```
char array[]="ab\
cd\
ef";
```

3.5 The *void* data type

Functions that don't return a value can be declared to return a *void*. This provides a safety check on the use of such functions: if a *void* function attempts to return a value, or if a function tries to use the

value returned by a *void* function, the compiler will generate an error message.

Variables can be declared to point to a *void*, and functions can be declared as returning a pointer to a *void*.

Unlike other pointers, a pointer to a *void* can be assigned to a pointer to any type of object, and vice versa. For other types of pointers, the compiler will generate a warning message if an attempt is made to assign one pointer to another, when the types of objects pointed at by the two pointers differ.

That is, the compiler will generate a warning message for the assignment statement in the following program:

```
main()
{
    char *cp;
    int *ip;
    ip = cp;
}
```

The compiler won't complain about the following program:

```
main()
{
    char *cp;
    void *getbuf();
    cp = getbuf();
}
```

3.6 Special symbols

Aztec C supports the following symbols:

___FILE___	Name of the file being compiled. This is a character string.
___LINE___	Number of the line currently being compiled. This is an integer.
___FUNC___	Name of the function currently being compiled. This is a character string.

In case you can't tell, these symbols begin and end with two underscore characters.

For example,

```
printf("file= %s\n", ___FILE___);
printf("line= %d\n", ___LINE___);
printf("func=%s\n", ___FUNC___);
```

3.7 String merging

The compiler will merge adjacent character strings. For example,

```
printf("file=" __FILE__ " line= %d func= " __FUNC__,
        __LINE__);
```

3.8 Long names

Symbol names are significant to 31 characters. This includes external symbols, which are significant to 31 characters throughout assembly and linkage.

3.9 Reserved words

const, *signed*, and *volatile* are reserved keywords, and must not be used as symbol names in your programs.

3.10 Global variables

The standard C language specifies that to access a global variable, exactly one module must declare it without the *extern* keyword and all others declare it with the *extern* keyword. Aztec C supports the following modified version of the rule:

- Multiple modules can declare the same variable, with the *extern* keyword being optional;
- When several modules declare a variable without using the *extern* keyword, the amount of space reserved for the variable is set to the largest size specified by the various declarations;
- When one module declares a variable using the *extern* keyword, at least one other module must declare the variable without using the *extern* keyword;
- At most one module can specify an initial value for a global variable;
- When a module specifies an initial value for a global variable, the amount of storage reserved for the variable is set to the amount specified in the declaration that specified an initial value, regardless of the amounts specified in the other declarations.

3.11 Data formats

3.11.1 char

Variables of type *char* are one byte long, and can be signed or unsigned. By default, a *char* variable is signed.

When a signed char variable is used in an expression, it's converted to a 16-bit integer by propagating the most significant bit. Thus, a char variable whose value is between 128 and 255 will appear to be a negative number if used in an expression.

When an unsigned char variable is used in an expression, it's converted to a 16-bit integer in the range 0 to 255.

A character in a *char* is in ASCII format.

3.11.2 pointer

Pointer variables are four bytes long.

3.11.3 short

Variables of type *short* are two bytes long. They can be signed or unsigned, and by default are signed.

A negative value is stored in two's complement format. A *short* is stored in memory with its least significant byte at the highest numbered address. A -2 stored at location 100 would thus look like:

location	*contents in hex*
100	FF
101	FE

3.11.4 long

Variables of type *long* occupy four bytes, and can be signed or unsigned.

Negative values are stored in two's complement format. Longs are stored sequentially with the most significant byte stored at the lowest memory address and the least significant byte at the highest memory address.

3.11.5 int

int variables are normally 16 bits long, but are 32 bits long if the *+L* compiler option is used. For more information, see the discussion of the +L option in the Options section of this chapter.

3.11.6 float & double

float and *double* numbers are both represented using IEEE format, occupying respectively 4 and 8 bytes of storage.

3.12 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
  ...
```

3.13 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.

3.13.1 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.

3.13.2 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- and 32-bit implementations do. 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;
```

3.13.3 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 processor on which the compiler-generated code will run. These symbols, and their corresponding processors, are:

symbol	*processor*
MPU8086	8086/8088
MPU80186	80186/80286
MPU6502	6502
MPU8080	8080
MPUZ80	Z80
MCH_AMIGA	Amiga
MCH_MACINTOSH	Macintosh
MCH_ATARI_ST	Atari ST
MCH_ROM	68000 Rom system

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 MACINTOSH
   /* Macintosh code */
#else
#ifdef MPU8086
   /* 8086 code */
#else
#ifdef MPU8080
   /* 8080 code */
#endif
#endif
#endif
```

4. Error checking

Compiler errors come in two varieties-- fatal and not fatal. Fatal errors cause the compiler to make a final statement and stop. Running out of memory and finding no input are examples of fatal errors. Both kinds of errors are described in the *Errors* chapter. The non-fatal sort are introduced below.

The compiler will report any errors it finds in the source file. It will first print out a line of code, followed by a line containing the up-arrow (caret) character. The up-arrow in this line indicates where the compiler was in the source line when it detected the error. The compiler will then display a line containing the following:

* The name of the source file containing the line;
* The number of the line within the file;
* An error code;
* The symbol which caused the error, when appropriate.

The error codes are defined and described in the *Errors* chapter.

The compiler writes error messages to its standard output. Thus, error messages normally go to the console, but they can be associated with another device or file by redirecting standard output in the usual manner. For example,

c68 prog	errors sent to the console
c68 prog >outerr	errors sent to the file *outerr*

The compiler normally pauses after every fifth error, and sends a message to its standard output asking if you want to continue. The compiler will continue only if you enter a line beginning with the character 'y'. If you don't want the compiler to pause in this manner, (if, for example, the compiler's standard output has been redirected to a file) specify the *-B* option when you start the compiler.

The compiler is not always able to give a precise description of an error. Usually, it must proceed to the next item in the file to ascertain that an error was encountered. Once an error is found, it is not obvious how to interpret the subsequent code, since the compiler cannot second-guess the programmer's intentions. This may cause it to flag perfectly good syntax as an error.

If errors arise at compile time, it is a general rule of thumb that the very first error should be corrected first. This may clear up some of the errors which follow.

The best way to attack an error is first to look up the meaning of the error code in the back of this manual. Some hints are given there as to what the problem might be. And you will find it easier to understand the error and the message if you know why the compiler produced that particular code. The error codes indicate what the compiler was doing when the error was found.

THE ASSEMBLER

Chapter Contents

The Assembler

The *as68* assembler translates assembly language source statements into relocatable object code. Assembler source statements are read from an input text file and the object code is written to an output file. A listing file is written if requested. The relocatable object code must be linked by *ln68*, the Manx Linker, before it can be executed. At linkage time it may be combined with other object files and run time library routines from system or private libraries. Object modules produced from C source text and Assembler source text can be combined at linkage time into a composite module.

Assembly language routines are generally not required when programming in C. Assembly language routines should only be necessary where critical execution time or critical size requirements exist. Some system interfacing or low level routines may also require assembler code.

Information on the MC68000 architecture and instructions can be found in the *Motorola MC68000 16-bit Microprocessor User's Manual* (Prentice-Hall, Inc., Englewood Cliffs, N. J. 07632)

1. Operating Instructions

The assembler is started by entering the command line:

```
as68 [-options] filename
```

where *[-options]* specify optional parameters and *filename* is the name of the file to be assembled.

The assembler reads assembly source statements from the input file, writes the translated relocatable object code to an output file, and if requested writes a listing to an output file. The assembler also will merge assembly code from other files on encountering an *include* directive.

1.1 The Input File

Specification of the extension on the source file name is optional: if not given, it's assumed to be *.asm*. For example the following command assembles the file *io.asm*:

```
as68 io
```

1.2 The Object Code File

The object code produced by the assembler is written to a file. By default, this file is placed in the directory that contains the source file,

and its name is derived from that of the input file by changing the extension to *.r*.

To write the object code to another file, use the *-o* option. For example, the following command assembles the source that's in *prog.asm*, sending the object code to the file *new.obj*. This latter file is placed in the current directory, since the *-o* option didn't specify otherwise.

```
as68 -o new.obj prog.asm
```

1.3 Listing File

If the *-L* option is specified, the assembler will produce a listing file with the same root as the input file and a filename extension of *.lst*. The listing file displays the source statements and their machine language equivalent. The listing also indicates the relative displacement of each machine instruction.

1.4 Optimizations

The assembler by default performs some optimizations on an assembly language source file, making just two passes through the assembly source file. Optimization can be disabled using the *-N* option; this causes the asssembler to run faster, since it makes just a single pass through the source and since it needn't optimize the code, but it makes the resultant code larger and slower.

The instructions affected by these optimizations are:

branches	Long branches are converted to short if possible, and branches to the following location will be deleted.
movem	If there are no registers, the instruction is deleted. If there is only one register, the shorter *move* instruction is substituted.
jsr	*bsr* is substituted if possible.

To make these optimizations, the assembler uses a dynamically-allocated table. If this table is filled, the assembler will continue, will generate correct, but not completely optimized, object code, and will tell you the number of additional entries that it could have used. You can then reassemble the module using the *-S* option to define a different table size.

1.5 Searching for *include* Files

By default the assembler searches just the current directory for files specified in *include* statements. Using the *-I* option and the INCL68 environment variable, you can make the assembler also search other directories for such files, thus allowing program source files and header files to be contained in different directories.

If the file name on the *include* directive specifies a directory or a drive name, the assembler will automatically search just the specified directory for the file.

1.5.1 The -I option

The *-I* option defines a single directory to be searched for a file specified in an *include* statement. The path descriptor follows the *-I*, with no intervening blanks. For example, the specification

```
as68 -i/db/include prog1
```

directs the assembler to search the */db/include* directory when looking for an *include* file.

Multiple *-I* options can be specified when the assembler is started, if desired, thus defining multiple directories to be searched.

1.5.2 The INCL68 Environment Variable

The INCL68 environment variable also defines areas to be searched for include files. The value of the variable consists of the names of the directories to be searched, with each pair of names separated by semicolons.

The command that is used to set environment variables varies from system to system. For example, on PCDOS the following command sets INCL68 so that the directory \ram\include is searched:

```
set INCL68=\ram\include
```

1.5.3 Include Search Order

When the assembler encounters an *include* statement, it searches directories for the file specified in the statement in the following order:

1. The current directory is searched.
2. The directories specified in the *-I* options are searched, in the order listed on the line that started the assembler;
3. The directories specified in the INCL68 environment variable are searched, in the order listed.

2. Assembler Options

2.1 Summary of options

-O *filename*	Send object code to *filename*.
-I*area*	Defines an area to be searched for files specified in an *include* statement.
-L	Generate listing.
-N	Don't optimize object code.
-S*num*	Create squeeze table having *num* entries.
-V	Verbose option. Generate memory usage statistics.
-ZAP	This option is used primarily when you assemble a file that was generated by the compiler. It directs the assembler to delete the input file after processing.
-C	Make *large code* the default code memory model. If this option isn't specified, *small code* is the default code memory model. The *near code* and *far code* directives can be used by a program to override the default code memory model.
-D	Make *large data* the default data memory model. If this option isn't specified, *small data* is the default data memory model. The *near data* and *far data* directives can be used by a program to override the default data memory model. For more information on memory models, see the Compiler chapter. For more information on the *near* and *far* directives, see the Programmer Information section of this chapter.
-E*name[=val]*	Create an entry in the symbol table for *name* and assign it the constant value *val*. If *val* isn't specified, *name* is assigned the value 1.

2.2 Description of options

2.2.1 The '-O *filename*' option

This option causes *as* to send the object code to *filename*. If this option isn't specified, *as* sends the object code to a file whose name is derived from that of the assembler source file by changing the extension to *.r*; in this case, the file is placed in the directory containing the source file.

2.2.2 The -I Option

The *-I* option causes the assembler to search in a specified area for files included in the source code.

The name of the area immediately follows the -I, with no intervening spaces. For example, the following defines directory /source/inc:

```
-I/source/inc
```

For more details, see the Assembler Operating Instructions, above.

2.2.3 The -L option

Causes *as68* to generate a listing. The name of the file to which the listing is sent is derived from that of the source file by changing the extension to *.lst*. The listing file is placed in the directory containing the source file.

2.2.4 The -S option

The *-S* option defines the number of entries in the squeeze table. If this option isn't specified, the table contains 1000 entries.

The number of entries immediately follows the *-S*, with no intervening spaces. For example, the following option tells the assembler to use a squeeze table containing 1050 entries:

```
-s1050
```

3. Programmer Information

The following sections discuss the four types of assembly language statements:

1. Comments
2. Instructions
3. Directives
4. Macro Calls

3.1 Comments

A comment can appear after a semicolon or after the operand field. For example:

```
; this is a comment
    link a6,#.2   this is also a comment
```

3.2 Executable Instructions

Executable instructions have the general format:

```
label operation operand
```

3.2.1 Labels

Assembler labels can be any length. External labels are only significant for the first 32 characters. Any additional characters will be ignored. Valid label characters include letters, numbers, or the special characters . and _. A label cannot begin with a digit.

Labels that do not start in the first column require a colon suffixed.

3.2.2 Operations

The assembler recognizes all of the mnemonics found in Motorola's *16-bit Microprocessor User's Manual.*

To specify a length for instructions which support multiple lengths, it is sufficient to suffix the instruction mnemonic with:

.B	Specifies a length of one byte
.W	Specifies a length of 16-bits
.L	Specifies a length of 32-bits

3.2.3 Operands

The operand field consists of one expression, or two expressions separated by a comma with no imbedded spaces. An expression is comprised of register mnemonics, symbols, constants, or arithmetic combinations of symbols or constants.

3.2.3.1 Symbols

Symbols or labels represent relocatable or absolute values. An absolute value is one whose value is known at assembly time. A relocatable value is one whose value is not known until the program is

actually loaded into memory for execution.

Relocatable expressions can only be expressed arithmetically as sums or differences. The difference between two relocatable expressions is absolute. The result of summing two relocatable expressions is undefined.

3.2.3.2 Constants

There are five type of constants: octal, binary, decimal, hexadecimal and string.

* An octal constant is expressed as an @ followed by a string of digits from the set 0 through 7 such as *@123* or *@777*.
* A binary constant is expressed as a % followed by a string of ones and zeroes such as *%10101* or *%11001100*.
* A decimal constant is a string of numbers.
* A hexadecimal constant is a *$* followed by a string of characters made up of numbers or alphabetics from *a* through *f* such as *$ffff* or *1a2e*.
* A string constant is any string of characters enclosed in single quotes such as *'abdc'*.

3.2.3.3 Registers

Register mnemonics are:

Name	*Register*
D0, ..., D7	Data registers
A0, ..., A7	Address registers
SP or *A7*	Stack pointer
PC	Program counter (forces PC relative mode)
SR	Status register
CCR	Condition code register
USP	User stack pointer

3.2.3.4 Operand expressions

The assembler supports operand expressions that use the following operators:

Operator	*Meaning*
+	Addition
-	Subtraction & unary minus
*	Multiplication
/	Division
>>	Shift right
<<	Shift left
&	And
|	Or

The order of precedence is innermost parenthesis, unary minus, shift, and/or, multiplication/division, and addition/subtraction.

3.3 Directives

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

EQU

label equ <expression>

This directive assigns the value of the expression on the right to the label on the left.

REG

label reg <register list>

This directive assigns the value of the register list to the label. Forward references are not allowed. A register list consists of a list of register names separated by the / character. The - character may be used to identify an inclusive set of registers. The following are valid register lists:

a0-a3/d0-d2/d4
a1/a2/a4/a6/d0-d2

PUBLIC

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

This directive identifies the specified symbols as having external scope. These symbols are visable to the linker and are used to resolve references between modules. The type of the symbol is CODE if it was defined within the code segment, DATA if it was defined within the data segment, and ABS if it was defined to have an absolute value in an *equ* directive.

GLOBAL and BSS

[label] global <symbol>,<size>
[label] bss <symbol>,<size>

These directives reserve storage for uninitialized data items. The area is reserved in the uninitialized data area. If *global* is used then the data item is known to other modules that are external to the routine. If *bss* is used then the data item is local to the routine in which it is defined.

If a *global* is defined in more than one module then the linkage editor will reserve the maximum value of those assigned.

A symbol that appears in both a *global* and a *public* directive is located in the initialized data area and the global statements size parameters are ignored.

ENTRY

[label] entry <symbol>

This directive defines the entry point of the program. Only one entry can be declared per program. If no entry point is defined, the first instruction of the first module becomes the default entry point.

END

This directive defines the end of the source statements. All files are closed and the assembler terminates.

CSEG

Assembled output following this directive is output into the code segment of the program output file.

DSEG

Assembled output following this directive is placed in the initialized data segment of the program file.

DC - Define Constant

[label]	*dc.b*	*<value>[,<value>, <value> ...]*
[label]	*dc*	*<value>[,<value>, <value> ...]*
[label]	*dc.w*	*<value>[,<value>, <value> ...]*
[label]	*dc.l*	*<value>[,<value>, <value> ...]*
[label]	*dc.b*	*"string"*

The *dc* directive causes one or more fields of memory to be allocated and initialized.

Each <value> operand causes one field to be allocated and then to be initialized with the specified value. A <value> can be an expression. An expression may contain forward references.

For command programs, a value can contain a reference to a memory location whose address won't be known until the program is loaded into memory. In this case, an item for this value will be added to the program's relocation table; when the program is loaded, the field containing this value will be set to the correct value.

Each field for a particular *dc* directive is the same length. A period followed by b, w, or l can be appended to a directive, defining the field length to be one, two, or four bytes, respectively. If the field length isn't specified in this way, it defaults to 2 bytes.

Fields that are two or four bytes long are aligned on word boundaries.

The last form listed above for *dc* allocates a field having exactly the number of characters in the string, and places the string in it.

DCB - Define Constant Block

[label]	*dcb.b*	*<size>[,<value>]*
[label]	*dcb*	*<size>[,<value>]*
[label]	*dcb.w*	*<size>[,<value>]*
[label]	*dcb.l*	*<size>[,<value>]*

The *dcb* directive allocates a block of storage containing <size> fields, and initializes each field with <value>. If <value> isn't specified, it's assumed to be 0.

Each field for a particular *dcb* directive is the same length. A period followed by b, w, or l can be appended to a directive, defining the field length to be one, two, or four bytes, respectively. If the field length isn't specified in this way, it defaults to 2 bytes.

Fields that are two or four bytes long are aligned on word boundaries.

DS - Define Storage

[label]	*ds.b*	*<size>*
[label]	*ds*	*<size>*
[label]	*ds.w*	*<size>*
[label]	*ds.l*	*<size>*

This directive allocates a block of storage containing <size> fields, and sets each field to 0.

Each field for a particular *ds* directive is the same length. A period followed by b, w, or l can be appended to a directive, defining the field length to be one, two, or four bytes, respectively. If the field length isn't specified in this way, it defaults to 2 bytes.

Fields that are two or four bytes long are aligned on word boundaries.

NEAR and FAR

near *code/data*
far *code/data*

The *near code* and *far code* directives cause the assembler to generate code that uses the *small code* or *large code* memory model, respectively. If these options aren't specified, the assembler will generate code whose code memory model is determined by the presence or absence of the +C assembler option.

The *near data* and *far data* directives cause the assembler to generate code that uses the *small data* or *large data* memory model, respectively. If these options aren't specified, the assembler will generate code whose data memory model is determined by the presence or absence of the +D assembler option.

A program can contain multiple *near* and *far* directives, thus allowing different sections of the same module to use different memory models.

LIST and NOLIST

The directives *list* and *nolist* turn on and off, respectively, the listing of assembly language statements to the listing file.

MLIST and NOMLIST

The directives *mlist* and *nomlist* specify whether or not the assembly language statements generated by a macro expansion should be written to the listing file.

CLIST and NOCLIST

The directives *clist* and *noclist* specify whether or not statements should be included in the listing file, when the statements were not assembled as a result of assembler conditional statements. By default, such statements are not listed.

INCLUDE

include *<file>*

This directive causes the assembler to suspend assembly of the current file and to assemble the specified file. When done, the assembler continues assembling the original file.

MACRO and ENDM

[label] macro <symbol>
...
text
...
endm

The specified symbol is entered in the assembler opcodes table. The text between the *macro* and *endm* is saved in memory. When the macro symbol is encountered as an opcode the text is placed in line. Up to nine arguments can be specified. They are referenced in the macro text as %1 through %9. In expanding a macro symbolic argument references are replaced by their actual value.

MEXIT

Upon encountering this directive expansion of the current macro stops and the assembler scans for the statement following the ENDM directive.

IF, ELSE, and ENDC

if <test>
...
[else]
...
endc

These directives are used to allow conditional assembly of parts of the input file. The general form of the IF test is:

<exp>		
<exp> == <exp>	||	*<exp> = <exp>*
<exp> != <exp>	||	*<exp> <> <exp>*
'str1' == 'str2'	||	*'str1' = 'str2'*
'str1' != 'str2'	||	*'str1' <> 'str2'*

If the test result is true, then the lines up to an *ELSE* or *ENDC* are assembled. If there is an *ELSE*, then lines up to the *ENDC* are skipped. The skipped lines are not displayed in the listing file unless the *CLIST* directive has been used. If the test is false, then lines are skipped until an *ELSE* or *ENDC* is encountered. If it is an *ELSE*, then the following lines up to an *ENDC* are assembled.

An undefined symbol is treated as having the value 0.

THE LINKER

Chapter Contents

The Linker

This chapter describes the *ln68* linker. It first gives a brief introduction to linking; the second and third sections give detailed operator-type information about the linker; and the fourth section gives programmer-type information.

1. Introduction to linking

C encourages modular programming; that is, the partitioning of a program into source modules that are separately compiled and assembled. The compilation and assembly of a source module generates an "object module". The linker links together all of a program's object modules, creating an executable program.

Programs typically consist of many object modules. Since it would be inconvenient to explicitly specify each module whenever you link a program, Aztec C68k/ROM supports object module libraries. When you pass a library's name to the linker, it examines the library's modules, and links into the program just those that are needed.

Aztec C68k/ROM provides source for several frequently-used functions, and for support routines that are called by compiler-generated code to perform operations such as arithmetic computation, etc. These are in source form, and part of the process of installing Aztec C68k/ROM is to compile and assemble them and then create object module libraries of them. In the following discussion, we refer to one of these libraries, *c68.lib*, which contains non-floating point functions, and whose modules have been compiled to use the small code, small data memory model.

Some of the provided functions, called "standard i/o" functions, perform high-level i/o by calling functions that you must write, as described in the Library Generation chapter. In the following discussion, we assume that you have implemented these functions, and thus that your *c68.lib* library supports the standard i/o function *printf*.

Creating the 'hello, world' program

Let's consider the creation of the "hello, world" program, whose main module, in the file *hello.c*, looks like this:

```
main()
{
    printf("hello, world\n");
}
```

The object modules that must be linked together include *hello.r*, the *printf* module from *c68.lib*, and other "support" modules from *c68.lib*. You don't explicitly generate calls to these support modules; they're automatically generated by the compiler. The command to link the program is:

```
ln68 hello.r -lc68
```

The *hello.r* operand causes the linker to include *hello.r* in the program. The *-lc68* operand causes the linker to search for needed modules in the *c68.lib* library that's located in the directory specified by the INCL68 environment variable and to include them in the program.

Another example

As another example, consider a program consisting of two of your own modules, plus whatever modules are needed from *c68.lib*. The source for the first of these modules, *file1.c*, looks like this:

```
main()
{
  printf("second example");
  func1();
  func2();
}

func1()
{
  return;
}
```

The source for the second module, *file2.c*, looks like this:

```
func2()
{
        return;
}
```

The command to link this program is:

```
ln68 file1.r file2.r -lc68
```

This causes the linker to include object modules *file1.r* and *file2.r* in the program, and to search for other needed modules in *c68.lib*.

Symbol reference and definition

As the linker proceeds, it keeps track of the global symbols that each module references and defines. For the linkage to succeed, each symbol that's referenced must also be defined; there can be multiple references to the same symbol.

Here are some examples of symbol reference and definition:

* A call of a function is a reference to that function's name;

- The actual definition of a function is a definition of the function's name;
- A variable declaration that includes the *extern* keyword is a reference to the variable.
- A global declaration of a variable that doesn't include the *extern* keyword is a definition of the variable.

For example, in the above sample program, *file1* contains references to *printf*, *func1*, and *func2*, and to support routines; it contains definitions of *main* and *func1*. *file2* contains a definition of *func2*, and references to support routines. Within *c68.lib* are modules that define *printf* and the support routines.

When the linker has examined all the modules that are going to be linked into a program, it checks its lists of defined and referenced symbols. If there are symbols that are referenced but not defined, the linker issues messages saying that those symbols are undefined and then halts without completing the linkage. For example, if the link command for the above program specified just *file1.r*, the linker would issue a message saying that *printf*, *func2*, and the support routines were undefined, since the references to those symbols were not matched by definitions. It doesn't say that *func1* is undefined, because the reference to it is matched by its definition in the same file.

Searching libraries

When the linker is searching a library, it checks each module's defined code symbols (ie, symbols that are defined in the module's code segment), looking for symbols that have been been referenced but not defined in the modules that have already been included in the program. If it finds such a symbol, it includes the module that contains it in the program. For example, in the above linkage the symbol *printf* is referenced but not defined when the linker begins searching *c68.lib*. When the linker looks at the library's module that contains the definition of the *printf* code symbol, the linker includes that module in the program it's building.

It's important to note that only the definition of a code segment symbol in a library module can cause the linker to include the module in a program. For example, in the above linkage the definition of a *printf* data symbol (ie, a symbol located in the data segment) in a library module would not cause the linker to include that module in the program.

The ordering of module and library names on the command line

The order in which modules and libraries are specified on the command line is important, since the linker processes files in this order.

For example, an attempt to link the "hello, world" program with the following command will fail:

```
ln68 -lc68 hello.r
```

For this command, the linker first scans *c68.lib* and then *hello.r*. When it scans *c68.lib* there aren't yet any referenced but undefined symbols, so the linker won't include any of the library's modules in the program. When it includes *hello.r* in the program, *printf* and the referenced support routines become referenced but undefined. But since *hello.r* is the last module specified on the command line, the linker won't go back and rescan *c68.lib*; so the undefined symbols remain undefined, and the linkage fails.

The moral of this is that it's good practice to leave all libraries at the end of the command line, with *c68.lib* at the very end.

The Order of Library Modules

For the same reason, the order of the modules within a library is significant, because the specification of a library on the command line causes the linker to search that library just once, from beginning to end. If a module is pulled in at any point, and that module introduces a new undefined symbol, then that symbol is added to the running list of undefined's. The linker will not search the library twice to find definitions for unmatched references.

For example, suppose you have a program that contains the modules *main.r*, *input.r*, *calc.r*, *output.r*, and any needed library modules, and that your modules have the following references:

module	*definitions*	*references*
main.r	main	in, calc
input.r	in	gets
calc.r	calc	out
output.r	out	printf

The command to link the program would look like this:

```
ln68 main.r input.r calc.r output.r -lc68
```

Suppose we build a library, *sub.lib*, to hold the last three modules of this program. Then our link step will look like this:

```
ln68 main.r -lsub -lc68
```

The order of the modules in *sub.lib* is important. For example, suppose *sub.lib*'s modules are in the following order:

```
input.r
output.r
calc.r
```

With the library in this order, here's how the above linkage would proceed:

1. The linker includes *main.r* in the program. After this step, *in* and *calc* are referenced but undefined (as are some other

symbols that are in *c68.lib*, but we're not concerned about them right now).

2. The linker begins searching *sub.lib*, and looks first at its *input* module. Since that module defines *in*, which is one of the linker's referenced but undefined symbols, it includes the *input* module in the program, takes *in* off its list of referenced but undefined symbols, and adds *gets* to it.
3. The linker looks at *output*, the next module in *sub.lib*. At this point, The symbols *calc* and *gets* are referenced but undefined. Since neither of these symbols are defined in *output*, the linker ignores it
4. The linker looks at *calc*, the next and last module in *sub.lib*. Since this module contains a definition of *calc*, one of the linker's referenced but undefined symbols, the linker includes *calc* in the program, removes *calc* from its list of referenced but undefined symbols, and adds *out* to the list.
5. The linker next scans *c68.lib*, and includes the modules within it that define *gets* and the support routines.

After scanning all of these modules and libraries, the *out* symbol is still referenced but undefined, so the linker will abort after logging the following message:

Undefined symbol: _out

This means that the module defining *out* was not pulled into the linkage. The reason, as we saw, was that *out* was not a referenced symbol when the linker scanned the *output* module, so the linker ignored it.

This problem would not occur if *sub.lib*'s modules were in the following order:

input.r
calc.r
output.r

The *ord68* library utility

The *ord68* utility simplifies the task of creating a library, by sorting a list of names of files that contain object modules. A library of these object modules that is created using the sorted list will be in the correct order.

There are some sets of object modules whose modules can't be put in a "correct" order; that is, for which it is impossible for the linker to decide which of the library's modules are needed by making just a single scan through the library. For such libraries, you can explicitly tell the linker to search the library multiple times.

For example, if *sub.lib* required two passes to find all needed modules, you could link the above program using the command

```
ln68 main.r -lsub -lsub -lc68
```

2. Using the Linker

The command to link a program looks like this:

ln68 [-options] file1.r [file2.r ...] [lib1.lib ...]

where *-options* are special options, *file1.r*, *file2.r* are names of the object modules that are to be included in the program, and *lib1.lib*, ... are names of the libraries that are to be searched for needed modules. The object modules must have been created using *as68* and the libraries by *lb68*.

The executable file

You can specify the name of the file to which the executable program is written with the -O linker option. Otherwise, the linker will derive the name of the output file from that of the first object module file listed on the command line, by deleting its extension. In the default case, the executable file will be located in the directory in which the first object file is located. For example,

ln68 prog.r -lc68

will produce the file *prog*, by linking the object module *prog.r* together with needed modules from the library *c68.lib*. (The -l option provides a convenient means of specifying libraries, as discussed below).

A different output file can be specified with the -O option, as in the following command:

ln68 -o program mod1.r mod2.r -lc68

Libraries

Source to many useful functions are provided with Aztec C68k/ROM, with which you can generate the libraries *c68.lib*, which contains the non-floating point functions, and *m68.lib*, which contains the floating point functions.

All programs must be linked with *c68.lib*. In addition to containing all the non-floating point functions described in the Functions chapter, it contains internal functions which are called by compiler-generated code.

Programs that perform floating point operations must be linked with *m68.lib* in addition to a version of *c68.lib*. The floating point library must be specified on the linker command line before *c68.lib*.

Libraries of your own modules can also be searched by the linker. These are created with the Manx *lb68* program, and must be listed on the linker command line before the Manx libraries.

For example, the following links the module *prog.r*, searching the libraries *mylib.lib*, *new.lib*, *m68.lib*, and *c68.lib* for needed modules:

```
ln68 program.r mylib.lib new.lib -lm68 -lc68
```

Each of the libraries will be searched once in the order in which they appear on the command line.

3. Linker Options

3.1 Summary of options

-O file Write executable code to the file named *file.*

-Lname Search the library *name.lib* for needed modules.

-F file Read command arguments from *file.*

-T Generate an ASCII symbol table file.

-V Be verbose; i.e. list detailed information about each segment.

+R dd Use address register *dd* for small model operations. *dd* is a decimal value, and default to 5 (ie, address register A5).

+C xxxx Set origin of code section to the hex value *xxxx* (default: 0).

+D xxxx Set origin of initialized data section to the hex value *xxxx* (default: immediately after the code section).

+U xxxx Set origin of the uninitialized data section to the hex value *xxxx* (default: immediately after the initialized data section).

+S xxxx Set the size of the stack area to the hex value *xxxx* (default: 2k).

+J xxxx Set the program's initial stack pointer to the hex value *xxxx.* (default: stack area immediately follows uninitialized data section, with size specified by *+S* option; stack pointer points to the top of this area).

+A Toggle 'long align' mode. When this mode is enabled, each module's code begins on a longword boundary; i.e. on a byte whose address is a multiple of 4. By default, this mode is disabled.

+Q Be quiet; i.e. don't list, on the console, each module that is included in a program. By default, the linker issues this list.

3.2 Detailed description of the options

The -O option

The *-O* option can be used to specify the name of the file to which the linker is to write the executable program. The name of this file is in the parameter that follows the *-O*. For example, the following command writes the executable program to the file *progout*:

```
ln68 -o progout prog.r -lc68
```

If this option isn't used, the linker derives the name of the executable file from that of the first input file, by deleting its extension.

The -L option

The *-L* option provides a convenient means of specifying to the linker a library that it should search, when the library is in a directory identified by the INCL68 environment variable, and when the extension of the library is *.lib*.

The name of the library is derived by concatenating the value of the environment variable CLIB68, the letters that immediately follow the -L option, and the string *.lib*. For example, with the libraries *subs.lib*, *io.lib*, *m68.lib*, and *c68.lib* in a directory specified by *CLIB68*, you can link the module *prog.r*, and have the linker search the libraries for needed modules by entering

```
ln68 prog.r -lsubs -lio -lm68 -lc68
```

The -F option

-F file causes the linker to merge the contents of the given file with the command line arguments. For example, the following command causes the linker to create an executable program in the file *myprog*. The linker includes the modules *myprog.r*, *mod1.r*, and *mod2.r* in the program, and searches the libraries *mylib.lib* and *c68.lib* for needed modules.

```
ln68 myprog.r -f argfil c68.lib
```

where the file *argfil*, contains the following:

```
mod1.r mod2.r
mylib.lib
```

The linker arguments in *argfile* can be separated by tabs, spaces, or newlines.

There are several uses for the *-F* option. The most obvious is to supply the names of modules that are frequently linked together. Since all the modules named are automatically pulled into the linkage, the linker does not spend any time in searching, as with a library. Furthermore, any linker option except *-F* can be given in a *-F* file. *-F*

can appear on the command line more than once, and in any order. The arguments are processed in the order in which they are read, as always.

The -T option

The *-T* option causes the linker to write the program's symbol table to a file. This file lists each of the program's symbols and its address. The file is organized into four sections:

1. Symbols in the code section (preceded by the line "Segment: 00 Hunk: 00);
2. Symbols in the initialized data section (preceded by the line "Segment: 00 Hunk: 01);
3. Symbols in the uninitialized data section, (preceded by the line "Segment: 00 Hunk: 02);
4. Values of the program's constant symbols (*STKSIZ* is the size of the program's stack area, and *__stkorg* is the initial stack pointer).

The symbol table file will have the same name as that of the file containing the executable program, with extension changed to *.sym*.

There are several special symbols which will appear in the table. They are defined later in this chapter, in the Programmer Information section.

The +R option

The *+R* option defines the address register that will be used in support of modules that use the small code and/or small data memory model. It has the format *+r dd*, where *dd* is the number of the address register.

For example, the following command tells the program to use address register A4 as the small model support register:

```
ln68 +r 4 main.r -lc68
```

If this option isn't specified, address register A5 is used.

If any of a program's modules use small code and/or small data, the small model support register points into the program's data sections. When a small data module attempts to access a variable that's in a data section, the variable's address is specified as a displacement from the small model support register. When a small code module calls a function that is more than 32k away from the call instruction, the linker will generate a jump instruction to the target function, place the instruction in the program's data area, and change the PC-relative call to a call of the generated jump instruction; the converted call will specify the address of the jump instruction as a displacement from the small model support register.

For more information about memory models, see the Programmer Information section of the Compiler chapter.

Options for positioning a program's sections

The linker organizes a program into three sections: code, initialized data, and uninitialized data. You can define the starting addresses of these sections using the +C, +D, and +U options; an option is followed by the hex value of the desired starting address.

By default, the code section begins at address 0, the initialized data section immediately after the code section, and the uninitialized data section immediately after the initialized.

For example, the following command creates the program *prog* whose code section begins at address 0, initialized data at 0x8000, and uninitialized data at 0x10000:

```
ln68 +d 8000 +u 10000 prog.o -lc68
```

Stack options

Two options affect a program's stack: +J and +S. The +J option defines the location at which the program's stack register initially points. The address in hex of this location follows the +J. For example, the following command creates a program whose stack register initially points at 0x20000:

```
ln68 +j 20000 prog.r -lc68
```

If the +J option isn't specified, the stack register will initially point to a location that follows the program's uninitialized data section. You can specify the distance between this location and the end of the uninitialized data section with the +S option. The hex value of the distance follows the +S. For example, the following command creates a program whose stack register initially points to a location that is 0x1000 bytes above the end of its uninitialized data section:

```
ln68 +s 1000 prog.r -lc68
```

The default value of the +S option is 2k; this means that when you specify neither the +S nor +J options, the program's stack register will point to a location that is 2k bytes beyond the end of its uninitialized data section.

The linker creates two stack-related symbols: *__Storg__*, whose value is the address initially pointed at by the linked program's stack register; and *STKSIZ*, whose value is the explicitly- or implicitly-defined value of the +S option. The standard startup routine uses *__Storg__* to set up the stack register; it doesn't use *STKSIZ*.

4. Programmer information

This section contains bits of information about the linker that you may find useful.

4.1 Program format

The linker creates a program that's in CP/M-68k format, with no relocation records.

4.2 Special linker-created symbols

When the linker creates a program, it defines several global symbols. These are:

__H0__org and *__H0__end*
: Beginning and ending addresses of program's code section.

__H1__org and *__H1__end*
: Beginning and ending addresses of program's initialized data section.

__H2__org and *__H2__end*
: Beginning and ending addresses of program's uninitialized data section.

__Storg__
: Initial contents of program's stack pointer.

STKSIZ
: Size of program's stack area (used when -J option isn't used).

4.3 Entry points

If a program has an "entry point", i.e. a symbol that's specified with the assembly language *entry* directive, and if the entry point isn't at the beginning of the program's code section, the linker will automatically create a jump to it at the beginning of the program's code section. Use of this feature can simplify the command line that passes instructions to the linker. However, this feature can't be used by programs whose code begins at location 0, since the first two words in memory must contain special information.

If you don't use this feature, you must explicitly define the startup routine's object module file to the linker, listing it first so that it is placed at the beginning of the program's code section. You must also explicitly use the -O option to define the name of the file to which the linker will write the created program, and not allow the linker to select the name (it would do so by taking the name of the first specified object module file and deleting its extension, which would result in all linked programs having the same name). Thus, if you don't use this feature, the simplest command line to the linker would be something like this:

```
ln68 -o prog /c68/lib/startup.r prog.r -lc68
```

If (1) you use this feature, (2) your entry point is also defined in a *public* directive, and (3) if your entry point is named *.begin*, you can place the startup routine's object module in a library, allowing the command line to the linker to be as simple as:

```
ln68 prog.r -lc68
```

Here's why the entry point must be named *.begin* and must be specified in a *public* directive: (1) when a module is compiled, the compiler automatically generates a reference to *.begin*; (2) when the linker is searching a library, these references are matched with the startup routine's definition of *.begin* in a *public* directive, and cause the linker to include the startup routine in the program. The presence of an *entry* directive in a library module doesn't cause the linker to automatically include that module in a program; it just identifies the specified symbol as being the entry point.

UTILITY PROGRAMS
for 68k/ROM TARGET SYSTEMS

Chapter Contents

Utility Programs for 68k/ROM Target Systems

This chapter describes the 68k/ROM-specific utility programs that are provided with this package. The host-specific utility programs are described in a separate chapter.

NAME

cnm68 - display object file info

SYNOPSIS

cnm68 [-sol] file [file ...]

DESCRIPTION

cnm68 displays the size and symbols of its object file arguments. The files can be object modules created by the Manx assembler, libraries of object modules created by the *lb* librarian, and, when applicable, 'rsm' files created by the Manx linker during the linking of an overlay root.

For example, the following displays the size and symbols for the object module *sub1.o* and the library *c.lib*:

```
cnm68 sub1.o c.lib
```

By default, the information is sent to the console. It can be redirected to a file or device in the normal way. For example, the following commands send information about *sub1.o* to the display and to the file *dispfile*:

```
cnm68 sub1.o
cnm68 sub1.o > dispfile
```

The first line listed by *cnm68* for an object module has the following format:

```
file (module): code: cc  data: dd  udata: uu  total: tt (0xhh)
```

where

- *file* is the name of the file containing the module,
- *module* is the name of the module; if the module is unnamed, this field and its surrounding parentheses aren't printed;
- *cc* is the number of bytes in the module's code segment, in decimal;
- *dd* is the number of bytes in the module's initialized data segment, in decimal;
- *uu* is the number of bytes in the module's uninitialized data segment, in decimal;
- *tt* is the total number of bytes in the module's three segments, in decimal;
- *hh* is the total number of bytes in the module's three segments, in hexadecimal.

If *cnm68* displays information about more than one module, it displays four totals just before it finishes, listing the sum of the sizes of the modules' code segments, initialized data segments, and uninitialized data segments, and the sum of the sizes of all segments of all modules. Each sum is in decimal; the total of all segments is also

given in hexadecimal.

The *-s* option tells *cnm68* to display just the sizes of the object modules. If this option isn't specified, *cnm68* also displays information about each named symbol in the object modules.

When *cnm68* displays information about the modules' named symbols, the *-l* option tells *cnm68* to display each symbol's information on a separate line and to display all of the characters in a symbol's name; if this option isn't used, *cnm68* displays the information about several symbols on a line and only displays the first eight characters of a symbol's name.

The *-o* option tells *cnm68* to prefix each line generated for an object module with the name of the file containing the module and the module name in parentheses (if the module is named). If this option isn't specified, this information is listed just once for each module: prefixed to the first line generated for the module.

The *-o* option is useful when using *cnm68* in combination with *grep*. For example, the following commands will display all information about the module *perror* in the library *c.lib*:

```
cnm68 -o c.lib >tmp
grep perror tmp
```

cnm68 displays information about an module's 'named' symbols; that is, about the symbols that begin with something other than a period followed by a digit. For example, the symbol *quad* is named, so information about it would be displayed; the symbol *.0123* is unnamed, so information about it would not be displayed.

For each named symbol in a module, *cnm68* displays its name, a two-character code specifying its type, and an associated value. The value displayed depends on the type of the symbol.

If the first character of a symbol's type code is lower case, the symbol can only be accessed by the module; that is, it's local to the module. If this character is upper case, the symbol is global to the module: either the module has defined the symbol and is allowing other modules to access it or the module needs to access the symbol, which must be defined as a global or public symbol in another module. The type codes are:

ab The symbol was defined using the assembler's EQU directive. The value listed is the equated value of its symbol.

The compiler doesn't generate symbols of this type.

pg The symbol is in the code segment. The value is the offset of the symbol within the code segment.

The compiler generates this type symbol for function names. Static functions are local to the function, and so have type *pg*; all other functions are global, that is, callable from other programs, and hence have type *Pg*.

dt The symbol is in the initialized data segment. The value is the offset of the symbol from the start of the data segment.

The compiler generates symbols of this type for initialized variables which are declared outside any function. Static variables are local to the program and so have type *dt*; all other variables are global, that is, accessable from other programs, and hence have type *Dt*.

ov When an overlay is being linked and that overlay itself calls another overlay, this type of symbol can appear in the rsm file for the overlay that is being linked. It indicates that the symbol is defined in the program that is going to call the overlay that is being linked.

The value is the offset of the symbol from the beginning of the physical segment that contains it.

un The symbol is used but not defined within the program. The value has no meaning.

In assembly language terms, a type of *Un* (the U is capitalized) indicates that the symbol is the operand of a *public* directive and that it is perhaps referenced in the operand field of some statements, but that the program didn't create the symbol in a statement's label field.

The compiler generates *Un* symbols for functions that are called but not defined within the program, for variables that are declared to be *extern* and that are actually used within the program, and for uninitialized, global dimensionless arrays. Variables which are declared to be *extern* but which are not used within the program aren't mentioned in the assembly language source file generated by the compiler and hence don't appear in the object file.

bs The symbol is in the uninitalized data segment. The value is the space reserved for the symbol.

The compiler generates *bs* symbols for static, uninitialized variables which are declared outside all functions and which aren't dimensionless arrays.

The assembler generates *bs* symbols for symbols defined using the *bss* assembler directive.

Gl The symbol is in the uninitialized data segment. The value is the space reserved for the symbol.

The compiler generates *Gl* symbols for non-static, uninitialized variables which are declared outside all functions and which aren't dimensionless arrays.

The assembler generates *Gl* symbols for variables declared using the *global* directive which have a non-zero size.

NAME

hex68 - Intel hex code generator

SYNOPSIS

hex68 [-options] prog

DESCRIPTION

hex68 translates the program that's in the file named *prog*, and that was generated by the Aztec C68k/ROM linker, into Intel hex code. The program can then be burned into ROM by feeding the hex code into a ROM programmer. The hex code is written to one or more files, each of which contains the hex code for one ROM chip.

The ROM chips that are generated from the *hex68* output files will contain the program's code, followed by a copy of its initialized data.

Note: when a ROM system is started, its RAM contains random values; the Aztec C68k/ROM startup routine sets up its initialized data area, using the copy that's in ROM.

hex68 assumes that the size of each ROM chip is 2 kb. You can explicitly define the size of each ROM using *hex68*'s *-P* option.

The output files: even- and odd-addressed bytes in the same chips

hex68 can optionally generate hex code so that the program's even-addressed bytes are in one set of ROM chips, and its odd-addressed bytes are in another. We'll discuss this option below. In this section we discuss the output files that are created when this option isn't used; i.e. when a program's even- and odd-addressed bytes are in the same set of ROM chips.

When neither -E nor -O is specified, *hex68* derives the name of each output file from that of the input file, by appending an extension of the form *.hnn*, where *nn* is a number. For example, if the name of the linker-generated file is *prog*, then the names of the output files generated by *hex68* are *prog.h00*, *prog.h01*, and so on, where the *.h00* file contains the hex code for the lowest-addressed ROM, *.h01* the hex code for the next ROM, etc.

For example, suppose that *hex68* is creating Intel hex code for a program whose code and copy of initialized data will reside in three 2-kb ROMs that begin at location 0. Then *hex68* will create the following files:

prog.h00 Contains the Intel hex code for the ROM chip that occupies addresses 0-0x7ff;

prog.h01 Contains the hex code for the ROM that occupies 0x800-0xfff;

prog.h02 Contains the hex code for the ROM that occupies 0x1000-0x17ff.

The output files: even- and odd-addressed bytes in separate chips

To place a program's even-addressed bytes in one set of ROM chips and its odd-addressed bytes in another, you must run *hex68* twice: once using the -E option to generate the hex code for the chips that contain the even-addressed bytes, and once using the -O option to generate hex code for the chips that contain the odd-addressed bytes.

When either -E or -O is specified, *hex68* generates one or more files, each of which contains the Intel hex code for one ROM chip. By default, the size of each chip is 2k bytes, but you can use the -P option to explicitly define the chip size.

When the -E option is specified, the extension of the files are of the form *.enn*, where *nn* is a decimal number. The *.e00* file contains the hex code for the first of the ROM chips that contain even-addressed bytes, the *.e01* file contains the hex code for the second ROM chip, and so on.

When the -O option is specified, the extension of the files are of the form *.onn*, where *nn* is a decimal number. The *.o00* file contains the Intel hex code for the first of the ROM chips that contain odd-addressed bytes, the *.o01* file contains the hex code bytes for the second ROM chip, and so on.

The options

hex68 supports the following options:

-Bx The program begins *x* bytes into the first ROM chip, where *x* is a hexadecimal number. If this option isn't specified, the program begins at the beginning of the first ROM chip.

-E Output hex code for the program's even-addressed bytes.

-O Output hex code for the program's odd-addressed bytes.

-Pn The size of each ROM is *n* k-bytes, where *n* is a decimal number. If this option isn't specified, the size defaults to 2kb. For example, the following command specifies that each ROM chip is 64kb long:

```
hex68 -p64 exmpl
```

NAME

lb68 - object file librarian

SYNOPSIS

lb68 library [options] [mod1 mod2 ...]

DESCRIPTION

lb68 is a program that creates and manipulates libraries of object modules. The modules must be created by the Manx assembler.

This description of *lb68* is divided into three sections: the first describes briefly *lb68*'s arguments and options, the second *lb68*'s basic features, and the third the rest of *lb68*'s features.

1. The arguments to *lb68*

1.1 The *library* argument

When started, *lb68* acts upon a single library file. The first argument to *lb68* (*library*, in the synopsis) is the name of this file. The filename extension for *library* is optional; if not specified, it's assumed to be *.lib*.

1.2 The *options* argument

There are two types of *options* argument: function code options, and qualifier options. These options will be summarized in the following paragraphs, and then described in detail below.

1.2.1 Function code options

When *lb68* is started, it performs one function on the specified library, as defined by the *options* argument. The functions that *lb68* can perform, and their corresponding option codes, are:

function	*code*
create a library	(no code)
add modules to a library	-a, -i, -b
list library modules	-t
move modules within a library	-m
replace modules	-r
delete modules	-d
extract modules	-x
ensure module uniqueness	-u
define module extension	-e
help	-h

In the synopsis, the *options* argument is surrounded by square brackets. This indicates that the argument is optional; if a code isn't specified, *lb68* assumes that a library is to be created.

1.2.2 Qualifier options

In addition to a function code, the *options* argument can optionally specify a qualifier, that modifies *lb68*'s behavior as it is performing the requested function. The qualifiers and their codes are:

verbose	-v
silent	-s

The qualifier can be included in the same argument as the function code, or as a separate argument. For example, to cause *lb68* to append modules to a library, and be silent when doing it, any of the following option arguments could be specified:

```
-as
-sa
-a -s
-s -a
```

1.3 The *mod* arguments

The arguments *mod1*, *mod2*, etc. are the names of the object modules, or the files containing these modules, that *lb68* is to use. For some functions, *lb68* requires an object module name, and for others it requires the name of a file containing an object module. In the latter case, the file's extension is optional; if not specified, the *lb68* that's supplied with native Aztec C systems assumes that it's *.o*, and the *lb68* that's supplied with cross development versions of Aztec C assumes that the extension is *.r*. You can explicitly define the default module extension using the *-e* option.

1.4 Reading arguments from another file

lb68 has a special argument, *-f filename*, that causes it to read command line arguments from the specified file. When done, it continues reading arguments from the command line. Arguments can be read from more than one file, but the file specified in a *-f filename* argument can't itself contain a *-f filename* argument.

2. Basic features of *lb68*

In this section we want to describe the basic features of *lb68*. With this knowledge in hand, you can start using *lb68*, and then read about the rest of the features of *lb68* at your leisure.

The basic things you need to know about *lb68*, and which thus are described in this section, are:

* How to create a library
* How to list the names of modules in a library
* How modules get their names

- Order of modules in a library
- Getting *lb68* arguments from a file

Thus, with the information presented in this section you can create libraries and get a list of the modules in libraries. The third section of this description shows you how to modify selected modules within a library.

2.1 Creating a Library

A library is created by starting *lb68* with a command line that specifies the name of the library file to be created and the names of the files whose object modules are to be copied into the library. It doesn't contain a function code, and it's this absence of a function code that tells *lb68* that it is to create a library.

For example, the following command creates the library *exmpl.lib*, copying into it the object modules that are in the files *obj1.o* and *obj2.o*:

```
lb68 exmpl.lib obj1.o obj2.o
```

Making use of *lb68*'s assumptions about file names for which no extension is specified, the following command is equivalent to the above command:

```
lb68 exmpl obj1 obj2
```

An object module file from which modules are read into a new library can itself be a library created by *lb68*. In this case, all the modules in the input library are copied into the new library.

2.1.1 The temporary library

When *lb68* creates a library or modifies an existing library, it first creates a new library with a temporary name. If the function was successfully performed, *lb68* erases the file having the same name as the specified library, and then renames the new library, giving it the name of the specified library. Thus, *lb68* makes sure it can create a library before erasing an existing one.

Note that there must be room on the disk for both the old library and the new.

2.2 Getting the table of contents for a library

To list the names of the modules in a library, use *lb68*'s *-t* option. For example, the following command lists the modules that are in *exmpl.lib*:

```
lb68 exmpl -t
```

The list will include some ***DIR*** entries. These identify blocks within the library that contain control information. They are created and deleted automatically as needed, and cannot be changed by you.

2.3 How modules get their names

When a module is copied into a library from a file containing a single object module (that is, from an object module generated by the Manx assembler), the name of the module within the library is derived from the name of the input file by deleting the input file's volume, path, and extension components.

For example, in the example given above, the names of the object modules in *exmpl.lib* are *obj1* and *obj2*.

An input file can itself be a library. In this case, a module's name in the new library is the same as its name in the input library.

2.4 Order in a library

The order of modules in a library is important, since the linker makes only a single pass through a library when it is searching for modules. For a discussion of this, see the tutorial section of the Linker chapter.

When *lb68* creates a library, it places modules in the library in the order in which it reads them. Thus, in the example given above, the modules will be in the library in the following order:

```
obj1 obj2
```

As another example, suppose that the library *oldlib.lib* contains the following modules, in the order specified:

```
sub1  sub2  sub3
```

If the library *newlib.lib* is created with the command

```
lb68 newlib mod1 oldlib.lib mod2 mod3
```

the contents of the newly-created *newlib.lib* will be:

```
mod1  sub1  sub2  sub3  mod2  mod3
```

The *ord* utility program can be used to create a library whose modules are optimally sorted. For information, see its description later in this chapter.

2.5 Getting *lb68* arguments from a file

For libraries containing many modules, it is frequently inconvenient, if not impossible, to enter all the arguments to *lb68* on a single command line. In this case, *lb68*'s *-f filename* feature can be of use: when *lb68* finds this option, it opens the specified file and starts reading command arguments from it. After finishing the file, it continues to scan the command line.

For example, suppose the file *build* contains the line

```
exmpl obj1 obj2
```

Then entering the command

```
lb68 -f build
```

causes *lb68* to get its arguments from the file *build*, which causes *lb68* to create the library *exmpl.lib* containing *obj1* and *obj2*.

Arguments in a -f file can be separated by any sequence of whitespace characters ('whitespace' being blanks, tabs, and newlines). Thus, arguments in a -f file can be on separate lines, if desired.

The *lb68* command line can contain multiple *-f* arguments, allowing *lb68* arguments to be read from several files. For example, if some of the object modules that are to be placed in *exmpl.lib* are defined in *arith.inc*, *input.inc*, and *output.inc*, then the following command could be used to create *exmpl.lib*:

```
lb68 exmpl -f arith.inc -f input.inc -f output.inc
```

A -f file can contain any valid *lb68* argument, except for another -f. That is, -f files can't be nested.

3. Advanced *lb68* features

In this section we describe the rest of the functions that *lb68* can perform. These primarily involve manipulating selected modules within a library.

3.1 Adding modules to a library

lb68 allows you to add modules to an existing library. The modules can be added before or after a specified module in the library or can be added to the beginning or end of the library.

The options that select *lb68*'s add function are:

option	*function*
-b target	add modules before the module *target*
-i target	same as *-b target*
-a target	add modules after the module *target*
-b+	add modules to the beginning of the library
-i+	same as *-b+*
-a+	add modules to the end of the library

In an *lb68* command that selects the *add* function, the names of the files containing modules to be added follows the add option code (and the target module name, when appropriate). A file can contain a single module or a library of modules.

Modules are added in the order that they are specified. If a library is to be added, its modules are added in the order they occur in the input library.

3.1.1 Adding modules before an existing module

As an example of the addition of modules before a selected module, suppose that the library *exmpl.lib* contains the modules

obj1 obj2 obj3

The command

lb68 exmpl -i obj2 mod1 mod2

adds the modules in the files *mod1.o* and *mod2.o* to *exmpl.lib*, placing them before the module *obj2*. The resultant *exmpl.lib* looking like this:

obj1 mod1 mod2 obj2 obj3

Note that in the *lb68* command we didn't need to specify the extension of either the file containing the library to which modules were to be added or the extension of the files containing the modules to be added. *lb68* assumed that the extension of the file containing the target library was *.lib*, and that the extension of the other files was *.o*.

As an example of the addition of one library to another, suppose that the library *mylib.lib* contains the modules

mod1 mod2 mod3

and that the library *exmpl.lib* contains

obj1 obj2 obj3

Then the command

lb68 -b obj2 mylib.lib

adds the modules in *mylib.lib* to *exmpl.lib*, resulting in *exmpl.lib* containing

obj1 mod1 mod2 mod3 obj2 obj3

Note that in this example, we had to specify the extension of the input file *mylib.lib*. If we hadn't included it, *lb68* would have assumed that the file was named *mylib.o*.

3.1.2 Adding modules after an existing module

As an example of adding modules after a specified module, the command

lb68 exmpl -a obj1 mod1 mod2

will insert *mod1* and *mod2* after *obj1* in the library *exmpl.lib*. If *exmpl.lib* originally contained

obj1 obj2 obj3

then after the addition, it contains

obj1 mod1 mod2 obj2 obj3

3.1.3 Adding modules at the beginning or end of a library

The options *-b+* and *-a+* tell *lb68* to add the modules whose names follow the option to the beginning or end of a library, respectively. Unlike the *-i* and *-a* options, these options aren't followed by the name of an existing module in the library.

For example, given the library *exmpl.lib* containing

obj1 obj2

the following command will add the modules *mod1* and *mod2* to the beginning of *exmpl.lib*:

lb68 exmpl -i+ mod1 mod2

resulting in *exmpl.lib* containing

mod1 mod2 obj1 obj2

The following command will add the same modules to the end of the library:

lb68 exmpl -a+ mod1 mod2

resulting in *exmpl.lib* containing

obj1 obj2 mod1 mod2

3.2 Moving modules within a library

Modules which already exist in a library can be easily moved about, using the *move* option, *-m*.

As with the options for adding modules to an existing library, there are several forms of *move* functions:

option	*meaning*
-mb target	move modules before the module *target*
-ma target	move modules after the module *target*
-mb+	move modules to the beginning of the library
-ma+	move modules to the end of the library

In the *lb68* command, the names of the modules to be moved follows the 'move' option code.

The modules are moved in the order in which they are found in the original library, not in the order in which they are listed in the *lb68* command.

3.2.1 Moving modules before an existing module

As an example of the movement of modules to a position before an existing module in a library, suppose that the library *exmpl.lib* contains

obj1 obj2 obj3 obj4 obj5 obj6

The following command moves *obj3* before *obj2*:

lb68 exmpl -mb obj2 obj3

putting the modules in the order:

obj1 obj3 obj2 obj4 obj5 obj6

And, given the library in the original order again, the following command moves *obj6*, *obj2*, and *obj1* before *obj3*:

lb68 exmpl -mb obj3 obj6 obj2 obj1

putting the library in the order:

obj1 obj2 obj6 obj3 obj4 obj5

As an example of the movement of modules to a position after an existing module, suppose that the library *exmpl.lib* is back in its original order. Then the command

lb68 exmpl -ma obj4 obj3 obj2

moves *obj3* and *obj2* after *obj4*, resulting in the library

obj1 obj4 obj2 obj3 obj5 obj6

3.2.2 Moving modules to the beginning or end of a library

The options for moving modules to the beginning or end of a library are *-mb+* and *-ma+*, respectively.

For example, given the library *exmpl.lib* with contents

obj1 obj2 obj3 obj4 obj5 obj6

the following command will move *obj3* and *obj5* to the beginning of the library:

lb68 exmpl -mb+ obj5 obj3

resulting in *exmpl.lib* having the order

obj3 obj5 obj1 obj2 obj4 obj6

And the following command will move *obj2* to the end of the library:

lb68 exmpl -ma+ obj2

3.3 Deleting Modules

Modules can be deleted from a library using *lb68*'s *-d* option. The command for deletion has the form

lb68 libname -d mod1 mod2 ...

where *mod1*, *mod2*, ... are the names of the modules to be deleted.

For example, suppose that *exmpl.lib* contains

obj1 obj2 obj3 obj4 obj5 obj6

The following command deletes *obj3* and *obj5* from this library:

lb68 exmpl -d obj3 obj5

3.4 Replacing Modules

The *lb68* option 'replace' is used to replace one module in a library with one or more other modules.

The 'replace' option has the form *-r target*, where *target* is the name of the module being replaced. In a command that uses the 'replace' option, the names of the files whose modules are to replace the target module follow the 'replace' option and its associated target module. Such a file can contain a single module or a library of modules.

Thus, an *lb68* command to replace a module has the form:

lb68 library -r target mod1 mod2 ...

For example, suppose that the library *exmpl.lib* looks like this:

obj1 obj2 obj3 obj4

Then to replace obj3 with the modules in the files *mod1.o* and *mod2.o*, the following command could be used:

lb68 exmpl -r obj3 mod1 mod2

resulting in *exmpl.lib* containing

obj1 obj2 mod1 mod2 obj4

3.5 Uniqueness

lb68 allows libraries to be created containing duplicate modules, where one module is a duplicate of another if it has the same name.

The option *-u* causes *lb68* to delete duplicate modules in a library, resulting in a library in which each module name is unique. In particular, the *-u* option causes *lb68* to scan through a library, looking at module names. Any modules found that are duplicates of previous modules are deleted.

For example, suppose that the library *exmpl.lib* contains the following:

obj1 obj2 obj3 obj1 obj3

The command

lb68 exmpl -u

will delete the second copies of the modules *obj1* and *obj2*, leaving the library looking like this:

```
obj1    obj2    obj3
```

3.6 Extracting modules from a Library

The *lb68* option *-x* extracts modules from a library and puts them in separate files, without modifying the library.

The names of the modules to be extracted follows the *-x* option. If no modules are specified, all modules in the library are extracted.

When a module is extracted, it's written to a new file; the file has same name as the module and extension *.o*.

For example, given the library *exmpl.lib* containing the modules

```
obj1 obj2 obj3
```

The command

```
lb68 exmpl -x
```

extracts all modules from the library, writing *obj1* to *obj1.o*, *obj2* to *obj2.o*, and *obj3* to *obj3.o*.

And the command

```
lb68 exmpl -x obj2
```

extracts just *obj2* from the library.

3.7 The 'verbose' option

The 'verbose' option, *-v*, causes *lb68* to be verbose; that is, to tell you what it's doing.

This option can be specified as part of another option, or all by itself. For example, the following command creates a library in a chatty manner:

```
lb68 exmpl -v mod1 mod2 mod3
```

And the following equivalent commands cause *lb68* to remove some modules and to be verbose:

```
lb68 exmpl -dv mod1 mod2
lb68 exmpl -d -v mod1 mod2
```

3.8 The 'silence' option

The 'silence' option, *-s*, tells *lb68* not to display its signon message.

This option is especially useful when redirecting the output of a list command to a disk file, as described below.

3.9 Rebuilding a library

The following commands provide a convenient way to rebuild a library:

```
lb68 exmpl -st > tfil
lb68 exmpl -f tfil
```

The first command writes the names of the modules in *exmpl.lib* t the file *tfil*. The second command then rebuilds the library, using a arguments the listing generated by the first command.

The *-s* option to the first command prevents *lb68* from sendin information to *tfil* that would foul up the second command. Th names sent to *tfil* include entries for the directory blocks, ***DIR***, bu these are ignored by *lb68*.

3.10 Defining the default module extension.

Specification of the extension of an object module file is optional the *lb68* that comes with native development versions of Aztec C assumes that the extension is *.o*, and the *lb68* that comes with cros development versions of Aztec C assumes that it's *.r*. You ca explicitly define the default extension using the *-e* option. This optio has the form

```
-e .ext
```

For example, the following command creates a library; th extension of the input object module files is *.i*.

```
lb68 my.lib -e .i mod1 mod2 mod3
```

3.11 Help

The *-h* option is provided for brief lapses of memory, and wil generate a summary of *lb68* functions and options.

NAME

obd68 - list object code

SYNOPSIS

obd68 <objfile>

DESCRIPTION

obd68 lists the loader items in an object file. It has a single parameter, which is the name of the object file.

NAME

ord68 - sort object module list

SYNOPSIS

ord68 [-v] [infile [outfile]]

DESCRIPTION

ord68 sorts a list of object file names. A library of the object modules that is generated from the sorted list by the Manx object module librarian will have a minimum number of 'backward references'; that is, global symbols that are defined in one module and referenced in a later module.

Since the specification of a library to the linker causes it to search the library just once, a library having no backward references need be specified just once when linking a program, and a library having backward references may need to be specified multiple times.

infile is the name of a file containing an unordered list of file names. These files contain the object modules that are to be put into a library. If *infile* isn't specified, this list is read from *ord68*'s standard input. The file names can be separated by space, tab, or newline characters.

outfile is the name of the file to which the sorted list is written. If it's not specified, the list is written to *ord68*'s standard output. *outfile* can only be specified if *infile* is also specified.

The *-v* option causes *ord68* to be verbose, sending messages to its standard error device as it proceeds.

NAME

srec68 - Motorola S-record generator

SYNOPSIS

srec68 [-options] prog

DESCRIPTION

srec68 translates the program that's in the file named *prog*, and that was generated by the Aztec C68k/ROM linker, into Motorola S-records The program can then be burned into ROM by feeding the S-records into a ROM programmer. The S-records are written to one or more files, each of which contains the hex code for one ROM chip.

The ROM chips that are generated from the *srec68* output files will contain the program's code, followed by a copy of its initialized data.

Note: when a ROM system is started, its RAM contains random values; the Aztec C68k/ROM startup routine sets up its initialized data area, using the copy that's in ROM.

srec68 assumes that the size of each ROM chip is 2 kb. You can explicitly define the size of each ROM using *srec68*'s *-P* option.

The output files: even- and odd-addressed bytes in the same chips

srec68 can optionally generate S-records so that the program's even-addressed bytes are in one set of ROM chips, and its odd-addressed bytes are in another. We'll discuss this option below. In this section we discuss the output files that are created when this option isn't used; i.e. when a program's even- and odd-addressed bytes are in the same set of ROM chips.

When neither -E nor -O is specified, *srec68* derives the name of each output file from that of the input file, by appending an extension of the form *.mnn*, where *nn* is a number. For example, if the name of the linker-generated file is *prog*, then the names of the output files generated by *srec68* are *prog.m00*, *prog.m01*, and so on, where the *.m00* file contains the S-records for the lowest-addressed ROM, *.m01* the S-records for the next ROM, etc.

For example, suppose that *srec68* is creating S-records for a program whose code and copy of initialized data will reside in three 2-kb ROMs that begin at location 0. Then *srec68* will create the following files:

prog.m00 Contains the S-records for the ROM chip that occupies addresses 0-0x7ff;

prog.m01 Contains the S-records for the ROM that occupies 0x800-0xfff;

prog.m02 Contains the S-records for the ROM that occupies 0x1000-0x17ff.

The output files: even- and odd-addressed bytes in separate chips

To place a program's even-addressed bytes in one set of ROM chips and its odd-addressed bytes in another, you must run *srec68* twice: once using the -E option to generate the S-records for the chips that contain the even-addressed bytes, and once using the -O option to generate S-records for the chips that contain the odd-addressed bytes.

When either -E or -O is specified, *srec68* generates one or more files, each of which contains the S-records for one ROM chip. By default, the size of each chip is 2k bytes, but you can use the -P option to explicitly define the chip size.

When the -E option is specified, the extension of the files are of the form *.enn*, where *nn* is a decimal number. The *.e00* file contains the S-records for the first of the ROM chips that contain even-addressed bytes, the *.e01* file contains the S-records for the second ROM chip, and so on.

When the -O option is specified, the extension of the files are of the form *.onn*, where *nn* is a decimal number. The *.o00* file contains the S-records for the first of the ROM chips that contain odd-addressed bytes, the *.o01* file contains the S-records for the second ROM chip, and so on.

The options

srec68 supports the following options:

-An The size of an S-record's address field is *n* bytes, where (following Motorola specifications) *n* can be 2, 3, or 4. If this option isn't specified, the field size defaults to 2 bytes.

-Bx The program begins *x* bytes into the first ROM chip, where *x* is a hexadecimal number. If this option isn't specified, the program begins at the beginning of the first ROM chip.

-E Output S-records for the program's even-addressed bytes.

-O Output S-records for the program's odd-addressed bytes.

-Pn The size of each ROM is *n* k-bytes, where *n* is a decimal number. If this option isn't specified, the size defaults to 2kb. For example, the following command specifies that each ROM chip is 64kb long:

srec68 -p64 exmpl

LIBRARY GENERATION

Chapter Contents

Library Generation

The Aztec C68k/ROM functions are provided in source form. Before you can create programs that use them, you must make any necessary modifications to the library functions, and then create object module libraries of them.

We assume that you have installed Aztec C68k/ROM in a set of subdirectories, as directed in the Tutorial chapter. We also assume that your system has a *make* program maintenance program that is UNIX compatible; this program, under direction of "makefiles" provided with Aztec C68k/ROM, will control the compilation and assembly of library modules and the generation of the libraries. For systems whose standard software doesn't include *make*, we will provide the Aztec *make* with your Aztec C68k/ROM package, if one is available; otherwise, the release document will describe the procedure for creating the libraries.

The first section of this chapter discusses changes that you might make to the library functions. The second section discusses generation of the libraries.

The calling sequences (passed parameters, return values, error codes, etc) of most functions described in the first section are presented in the System Independent Functions chapter. The calling sequences for the other functions are appended to this chapter.

. Modifying the functions

Many of the functions provided with this package will run, without modification, on any 68000-based system. Some, however, may need to be modified for use on your system.

The functions that may need to be rewritten are:

a. The startup function;
b. The unbuffered i/o functions;
c. The standard i/o functions *agetc* and *aputc*;
d. The low-level heap allocation functions *brk* and *sbrk*;
e. The exit functions *exit* and *_exit*.

1.1 The startup function

A program's startup routine is executed when the program is started. It performs program initialization and then calls the program's *main* function.

The source for the startup routine that is provided with Aztec C68k/ROM is in the file *rom68.a68*, in the *rom68.arc* archive. The

supplied version of this routine makes the following assumptions about a program that contains it, and about a system that contains the program:

* The system's startup/reset vectors and interrupt vector table are in ROM.
* The program is the "startup program" of the system containing it. That is, the program will gain control on system startup or reset.
* The program's code and a copy of its initialized data are in ROM. It's the startup routine's duty to set up the program's initialized data area in RAM from the ROM copy.
* The startup routine is at the beginning of the program's code segment.
* The system doesn't support interrupts.

If these assumptions aren't satisfied by your system, you will have to modify the startup routine. The following paragraphs discuss changes that can be made for several types of programs and systems.

1.1.1 Startup routines for ROM-based 'startup programs' on interrupt-driven systems

Since a system's memory must begin with startup vectors and be followed by the table of interrupt vectors, the above assumptions mean that the startup module must contain assembly language statements that pre-initialize these vectors. In fact, the supplied startup routine does contain statements that pre-initialize the startup vectors: the stack vector points to the top of the area that's reserved for the stack, and the code vector points to the *.begin* label in the startup module.

However, the supplied startup routine can't pre-initialize the interrupt vector table, since that's system dependent. The supplied startup routine simply reserves space for the table.

Thus, if a program that satisfies all the above assumptions is to be placed on a system that supports interrupts, you must modify the startup routine, replacing the statement that reserves space for the interrupt table with statements that pre-initialize the vectors for supported interrupts.

1.1.2 Startup routines for ROM-based, non-startup programs

If the startup routine will be included in programs that will be burned into ROM but that won't be a system's startup program, you can remove the statements in the startup routine that pre-initialize the system startup vectors and that reserve space for the interrupt table.

Most of the code in a program's startup routine needs to be executed just once. For example, its initialized data area in RAM needs to be set up from the copy in ROM just once; and its uninitialized data area needs to be cleared just once. So if a program

will be called more than once, you could design your startup routine so that this special startup code is executed just once. The advantages to this are (1) it speeds up interprogram calls, (2) variables are preserved between interprogram calls.

To do this, you could have a second entry point into a program, in addition to the standard entry point. The first call is made to the standard entry point, and all subsequent calls are made to the secondary entry point.

The secondary entry point performs just those operations that need to be done on each entry to the program. For example, if the program uses the small code or small data memory model, the secondary entry point would save the contents of the small model support register and set it up for the called program.

To implement the two entry points, you could add two jump instructions to the beginning of the program's startup routine: the first jumps to the startup routine's *.begin* label; the second jumps to the secondary entry point code.

1.1.3 Startup routines for systems whose interrupt table is in RAM

The interrupt vector table of some 68k systems must reside in RAM, to enable the program to dynamically set up and change the vectors. Since this table is normally in ROM, this requires special hardware and corresponding changes to the startup routine.

In this section first we describe why the interrupt table normally resides in ROM. We then present two hardware techniques used to place the table in RAM and the corresponding changes that must be made to the startup routine.

1.1.3.1 Why the interrupt table is normally in ROM

On a 68k system, the startup vectors occupy the first eight bytes of memory and the interrupt table follows. If the system uses a standard configuration (i.e. a typical microcomputer system configuration that doesn't use special hardware), then the startup vectors must be in ROM, so that they will be already initialized when the system is turned on or reset. Since the smallest ROM chip is about 2K bytes, this in turn means that the interrupt table of a standard 68k system must also be in ROM.

1.1.3.2 Solution one: move the startup vectors

One way to allow the interrupt table to reside in RAM is to move the startup vectors away from the interrupt table:

a. Put RAM in the lowest-addressed section of memory, so that it extends at least from location 0 through the end of the interrupt table;

b. Include the startup vectors in the code section of the system's startup program, at a fixed offset from the beginning of the program's ROM;

c. Insert special hardware on the address bus between the processor and memory. On powerup or system reset, this hardware intercepts the processor's first two accesses of memory, which are requests by the processor for the startup vectors, and translates the accompanying addresses (i.e. locations 0 and 4) to those of the fields within ROM that actually contain the startup vectors.

To support this hardware configuration, you should remove the statement in the startup module that reserves space for the interrupt table and add executable code that initializes the table. To put the startup vectors at a fixed place in ROM memory, to which the special hardware can redirect attempts by the processor to access them, you could leave the statements that define the startup vectors in the startup routine and then link the startup routine as the program's first module. The startup vectors will then be in ROM, in the first eight bytes of the startup program's code section.

1.1.3.3 Solution 2: move the interrupt table

Another way to put the interrupt table in RAM is to move the interrupt table away from the startup vectors:

a. Put the RAM for the interrupt table in an unused section of the system's memory space, a section that is not near the low end of memory.

b. Put the ROM that contains the code for the system's startup program in memory, beginning at location 0. The startup routine should be at the beginning of the program's code section; the only changes that it needs are executable statements that initialize the interrupt table.

c. Put a programmable logic array on the address bus, between memory and the processor. This will intercept requests to access an interrupt vector (i.e. accesses of memory between locations 8 and 0x400) and translate the accompanying address to the address in RAM at which the vector is actually located.

1.1.4 Defining the heap

The startup routine initializes variables that define the boundaries of a program's heap, so that the heap occupies the 2k-byte area of memory that's just above the program's stack area. If your system's heap space isn't in this area, you will have to change these initializations.

These variables, which are used by the *sbrk* and *brk* functions, are:

_mbot Points at the bottom of the heap.
_mtop Points at the top of the heap.
_mcur Points at the top of allocated heap space.

These are the names that a C-language module uses to access the variables; an assembly language module uses these names, with an additional prepended underscore (e.g. *___mbot*).

1.1.5 ROM-based initialized data

The startup routine contains statements that set up a program's initialized data segment in RAM from its copy in ROM. Remove these statement if the program's initialized data is to remain in ROM; i.e. if you linked the program without using the linker's -D option.

1.1.6 Startup routines for RAM-based programs

If you are creating programs that won't be put into ROM (for example, programs that will run on a system that uses the CP/M-68k operating system), here are some changes you may want to make to the startup routine:

- Remove the code that initializes the startup vectors and that reserves space for the interrupt table.
- Change the code that sets up the stack register. The operating system probably defines the area reserved for a program's stack (for example, on entry the stack register may already be initialized). If it doesn't, you could, for example, define space for the stack in the uninitialized data area, and point the stack register at the top of this area.
- Remove the code that moves the copy of initialized data from ROM to RAM.
- Change the code that initializes the pointers to heap space.
- The startup routine jumps directly to the *main* function. If you want your system to support the passing of arguments to the *main* function, you may want to have the startup routine call a C-language function, which gets the arguments (for example, getting them from the console) and then calls *main*.

1.2 The Unbuffered i/o functions

There are two classes of UNIX-compatible i/o functions: standard and unbuffered. The unbuffered i/o functions are system dependent, and the standard i/o functions call the unbuffered. The unbuffered i/o functions that are in the Aztec C68k package are merely stubs; so you must write those that your functions call, and those that are called by the standard i/o functions that your functions call.

The unbuffered i/o functions are:

open	creat	close	read	write
lseek	rename	unlink	ioctl	isatty

Descriptions of the unbuffered i/o functions are in the "System Independent Functions" and "Library Functions Overview" chapters. The following paragraphs present additional information that may be of use when writing your own versions of these functions.

1.2.1 File descriptors

Associated with each file or device that is open for unbuffered i/o is a positive integer called a "file descriptor". A file descriptor is one of the parameters that is passed to an unbuffered i/o function; it defines the file or device on which the i/o is to be performed. There's usually a limited number of file descriptors, which of course limits the number of files and/or devices that can be simultaneously open for i/o.

1.2.1.1 When there's lots of files and devices...

If a system supports disk files and/or supports more devices than file descriptors, the file descriptors must be dynamically allocated. That is, before i/o with a file or device can begin, a function must be called that assigns a file descriptor to it; and when the i/o is done another function must be called to de-assign the file descriptor. In this case, a table is usually provided that has entries defining the status of each file descriptor and that is accessible to all the unbuffered i/o functions. Here's how the unbuffered i/o functions make use of the table:

* *open* and *creat* prepare a file or device for unbuffered i/o. They scan the table for an unused entry, and initialize the entry with information about the file or device. For example, the entry for an open device might contain the device's address; that for an open file might contain the file's current position and access mode. As the file descriptor for the opened file or device, *open* and *creat* return the entry's index into the table.

* *read*, *write*, *lseek*, *ioctl*, and *isatty* perform operations on, and determine the status of, an open file or device. The file descriptor of the file or device is one of the parameters passed to them. They examine the file descriptor's table entry for information about the file or device.

* *close* completes i/o to the open file or device having a specified file descriptor. Most of the operations that *close* performs depend on the particular file or device; but it always marks the descriptor's table entry as being unused.

* *unlink* and *rename* don't use the file descriptor table at all.

1.2.1.2 When only devices are supported...

If programs access just devices (i.e. not files), if there are fewer devices than file descriptors, and if your programs make limited use of the standard i/o functions (as defined below), you can simplify the unbuffered i/o functions by doing away with the file descriptor table, hard-coding the assignment of devices and file descriptors into the unbuffered i/o functions, and leaving *open*, *creat*, and *close* as mere stubs that simply return when called.

For example, you could code into the *write* function the fact that file descriptor 5 is associated with a printer at a certain address. Then to write to the printer, a program could simply issue a call to *write*, telling it to write to file descriptor 5. It wouldn't have to first call *open* or subsequently call *close*.

1.2.1.3 Pre-assigned file descriptors

By convention, file descriptors 0, 1, and 2 are pre-assigned to the system console, even when all other file descriptors are dynamically assigned. To perform an unbuffered i/o operation on the console, a program simply calls the appropriate function, specifying one of these file descriptors; it need not first call *open* or subsequently call *close*.

Some systems allow the operator to redirect file descriptors 0 and 1 to other files and/or devices, by specifying special operands on the command line that starts a program. This is done by inserting a special function between the startup routine and the user's *main* function. If any redirection operands are found in the command line, this special function closes the specified file descriptor by calling *close* and reopens it to the new file or device by calling *open*. By convention, the command line operand to redirect file descriptor 0 consists of "<" followed by the file or device name. The command line operand to redirect file descriptor 1 consists of ">" or ">>" followed by the file or device name. ">" causes a new file to be created. ">>" causes a file to be appended to, if it already exists, or to be created, if it doesn't exist.

1.2.2 Interaction of the standard i/o and unbuffered i/o functions

The standard i/o functions call the unbuffered i/o functions. Because of this, the standard i/o operations that a program will perform places implementation requirements on the unbuffered i/o functions. This section discusses those requirements, after first presenting general information on standard i/o file pointers and their relationship to unbuffered i/o file descriptors.

Before standard i/o can be performed on a file or device, an unbuffered i/o file descriptor must be assigned to it, and a standard i/o "file pointer" must be assigned to the file descriptor. The assignment of a file pointer and file descriptor can be done dynamically, by calling the standard i/o *fopen* function. Three file pointers, named stdin, stdout, and stderr, are pre-assigned to file

descriptors 0, 1, and 2; these file descriptors in turn are pre-assigned to the console.

When a program calls a standard i/o function, it often must pass a file pointer, which identifies the file or device on which i/o is to be performed. There are a special set of standard i/o functions for accessing stdin, stdout, and stderr: for these, the file pointer isn't passed, since the functions know what file pointer is being accessed.

1.2.2.1 Supporting the standard i/o *fopen* and *fclose* functions

The dynamic assignment of a file pointer and file descriptor to a file or device is done by the *fopen* function. This function selects a file pointer for the file or device and then calls the unbuffered i/o *open* function, which selects a file descriptor.

If programs call *fopen*, you must implement the unbuffered i/o *open* function, and *open* must return the file descriptor that's associated with the file or device. This requirement (for a functional *open* when *fopen* is called) must be met even if file descriptors are pre-assigned to devices; *open* in this case could be very simple, just searching a table for a device name and returning the associated file descriptor.

Conversely, the use of the standard i/o functions to access those devices that don't first have to be *fopen*ed (i.e. stdin, stdout, and stderr) places no requirements on *open*. In particular, if file descriptors are pre-assigned to devices and *open* simply returns when called, programs can still call the standard i/o functions to access the devices associated with the stdin, stdout, and stderr file pointers.

The standard i/o function *fclose* calls the unbuffered i/o function *close*. Thus, if programs call *fclose*, you must implement a *close* function. If assignments of devices to file descriptors is hard-coded, *close* can usually just return the value 0, since nothing special (such as calling the operating system to close an open file or deallocating a file descriptor) needs to be done.

1.2.2.2 Supporting the standard i/o input and output functions

If programs call any of the standard i/o input functions, you must implement the unbuffered i/o *read* function. And if they call any of the standard i/o output functions, you must implement the *write* function.

1.2.2.3 Supporting the standard i/o *fseek* function

If programs will call the standard i/o *fseek* function, you must implement the unbuffered i/o *lseek* function, since *fseek* calls *lseek*.

1.2.2.4 Standard i/o and the *isatty* function

If programs call any standard i/o functions, you must implement the unbuffered i/o function *isatty*. The standard i/o functions call this function to decide whether their i/o to a file or device should be

buffered or unbuffered.

This use of the word "unbuffered" in describing standard i/o might be a little confusing, since the use of the expression "unbuffered i/o functions" to describe one set of i/o functions implies that the other set, the "standard i/o functions", are buffered. Nevertheless, a standard i/o stream can be either buffered or unbuffered: if buffered, data that's exchanged between user-written functions and the unbuffered i/o functions passes through a buffer; if unbuffered, data doesn't pass through a buffer.

For a given file descriptor, *isatty* should return non-zero if standard i/o to the device associated with the file descriptor is to be buffered, and zero if it is to be unbuffered.

For example, *isatty* should probably return non-zero for a file descriptor that's associated with the system console and zero for file descriptors associated with files; it could return either zero or non-zero for other devices, such as printers, depending on your system's requirements.

1.2.3 Error codes

We've presented most of the factors you should consider when writing your unbuffered i/o functions. In this section we want to list error codes that the functions could return in the global *int errno*.

open error codes:

ENOENT	File does not exist and O_CREAT wasn't specified.
EEXIST	File exists, and O_CREAT+O_EXCL was specified.
EMFILE	Invalid file descriptor passed to *open*.

close error codes:

EBADF	Bad file descriptor passed to *close*.

creat error codes:

EMFILE	All file descriptors are in use.

lseek error codes:

EBADF	Invalid file descriptor
EINVAL	Offset parameter is invalid, or the requested position is before the beginning of the file.

read error codes:

EBADF	Invalid file descriptor

write error codes:

EBADF	Invalid file descriptor
EINVAL	Invalid operation; i.e. writing not allowed.

1.3 The standard i/o functions *agetc* and *aputc*

The characters used to terminate lines of text differ from system to system. On UNIX, it's the newline (linefeed) character, '\n'. On the Apple //, it's carriage return, '\r'. On CPM, it's carriage return-line feed. In order to allow programs to access files of text in a system-independent manner, the standard i/o functions *agetc* and *aputc* are provided: *agetc* reads a character from the standard input channel, translating the line termination sequence into '\n'. *aputc* writes a character to the standard output channel, translating '\n' to the line termination sequence.

The following standard i/o functions call *agetc* and *aputc*:

scanf	fscanf	printf	fprintf
getchar	gets	fgets	
putchar	puts	fputs	

The ROM versions of *agetc* and *aputc* assume that '\n' separates lines of text; if this isn't the case for your system, you may need to modify *agetc* and *aputc*.

The source for these functions are in the files *agetc.c* and *aputc.c*, within the *stdio.arc* archive. If you followed our recommendations for installing Aztec C68k/ROM, dearchived versions are also in the STDIO subdirectory of the LIB directory.

1.4 The *sbrk* and *brk* heap management functions

sbrk and *brk* provide an elementary means of allocating and deallocating space from a program's heap. *sbrk* is called by the more sophisticated heap-allocation functions (*malloc*, etc), and *malloc* is called by the standard i/o functions; thus, if your programs call *malloc* or the other high-level heap management functions, or if they call the standard i/o functions, you will need to write an *sbrk* function.

Descriptions of the calling sequences for *sbrk* and *brk* are appended to this chapter.

You probably won't have to modify *sbrk* or *brk*, since the most system-dependent code (which defines the boundaries of the heap) is in the startup routine. But if you do, here are some things you should consider:

* A buffer allocated by *sbrk* should be on a quad-byte boundary (i.e. the address of its first byte should be divisible by four), since words on a 68000 or 68010 must be on an even-byte boundary and since long words on a 68020 can be most efficiently accessed if they're on a quad-word boundary.
* *malloc* assumes that the heap is a single, contiguous section of memory: when told to allocate a large block of memory, *malloc* makes repeated calls to *sbrk* for small blocks of memory, and then attempts to coalesce the small blocks into

one large block.

1.5 The *exit* and *_exit* functions

exit and *_exit* are called to terminate the execution of a program. They aren't usually called by ROM-based programs, since such programs usually don't terminate.

They are called, however, by RAM-based programs that are running in an operating system environment, since these programs usually do terminate.

When these functions are needed, you will have to modify *_exit*, since it must return to the operating system. But you can probably use *exit* as is, since it closes open files and devices in a system-independent way and then calls *_exit*.

Descriptions of the calling sequences to *exit* and *_exit* are appended to this chapter.

2. Building the libraries

Once you've made modifications to the supplied library functions, you can build your libraries. We've provided *makefiles* (which give directions to the *make* program) and *lb68* command files that will make this task easier; they will make the following libraries:

c68.lib	General purpose functions (small code, small data memory model);
c68ll.lib	General purpose functions (large code, large data);
m68.lib	Floating point functions (small code, small data);
m68ll.lib	Floating point functions (large code, large data);

If you followed our recommendations for installing Aztec C68k/ROM, each of the LIB directory's subdirectories contains a makefile that causes *make* to compile and assemble the subdirectory's source files. There is a makefile in the LIB directory that will have *make* first generate each subdirectory's object modules and then make a library.

Before you can generate the libraries, you must do several things:

a. In each makefile, modify the rules that define how to convert a C source file to an object module, so that the command that starts the compiler uses the options that correctly define register usage on your system;

b. If you've written your own unbuffered i/o modules, you'll probably need to modify the makefile that's in the ROM68 directory;

c. In the LIB directory are the files *c68.bld* and *m68.bld*, each of which tells *lb68* how to create a library. *c68.bld* is used for generating *c68.lib* and *c68ll.lib*: modify these files if necessary.

d. The environment variable INCL68 must be set to the name of the INCLUDE directory; that is, to the name of the directory that contains the include files. The command to do this varies from system to system; on PCDOS, it's the *set* command.

e. If you have a RAM disk, you can speed up the library-generation process by defining it using the CCTEMP environment variable. For more information, see the description of CCTEMP in the Compiler chapter.

You are now ready to create the libraries. Set the default or current directory to the LIB directory and start *make*, passing to it the name of the library you want created. For example, to create *c68.lib*, you would enter:

```
make c68.lib
```

Once started, *make* will activate several other copies of *make*, each of which will compile and assemble the files in one of LIB's subdirectories; it will then start *lb68*, which will make the specified library from the object modules that are in the subdirectories, as directed by the appropriate *.bld* file.

At times during library generation, there will be two copies of *make* in memory, and another program. If your system doesn't have enough memory to hold all of these programs (in this case, *make* will abort with the message "EXEC failure"), it may still have enough to hold one copy of *make* and another program. In this case, you can create and execute batch files that will make the libraries. For each subdirectory, the batch file will first make that subdirectory the default or current directory; it will then activate *make*, using either the command *make* (to make small code, small data modules), or the command *make big* (to make large code, large data modules). The batch file will then activate *lb68*, passing to it the name of the appropriate *.bld* file.

3. Function descriptions

The System Independent Functions chapter presents the calling sequences of most of the functions that are discussed in this chapter. The remainder of this chapter presents the calling sequences of the other functions.

NAME

sbrk, brk

SYNOPSIS

brk(ptr)
void *ptr;

void *sbrk(size)

DESCRIPTION

sbrk and *brk* provide an elementary means of allocating and deallocating space from the heap. More sophisticated buffer management schemes can be built using these functions; for example, the standard functions *malloc*, *free*, etc call *sbrk* to get heap space, which they then manage for the calling functions.

sbrk increments a pointer, called the 'heap pointer', by *size* bytes, and, if successful, returns the value that the pointer had on entry. Initially, the heap pointer points to the base of the heap. *size* is a signed *int*; if it is negative, the heap pointer is decremented by the specified amount and the value that it had on entry is returned. Thus, you must be careful when calling *sbrk*: if you try to pass it a value greater than 32K, *sbrk* will interpret it as a negative number, and decrement the heap pointer instead of incrementing it.

brk sets the heap pointer to *ptr*, and returns 0 if successful.

SEE ALSO

The functions *malloc*, *free*, etc, implement a dynamic buffer-allocation scheme using the *sbrk* function. See the Dynamic Buffer Allocation section of the Library Functions Overview chapter for more information.

The standard i/o functions usually call *malloc* and *free* to allocate and release buffers for use by i/o streams. This is discussed in the Standard I/O section of the Library Functions Overview.

Your program can safely mix calls to the *malloc* functions, the standard i/o functions, and the *sbrk* and *brk* functions, as long as the calls to *sbrk* and *brk* don't decrement the heap pointer. Mixing *sbrk* and *brk* calls that decrement the heap pointer with calls to the *malloc* functions and/or the standard i/o functions is dangerous and probably shouldn't be done by normal programs.

ERRORS

If an *sbrk* or *brk* call is made that would result in the heap pointer passing beyond the end of the heap, *sbrk* and *brk* return -1, after setting the global integer *errno* to the symbolic value ENOMEM.

NAME

exit, _exit

SYNOPSIS

exit(code)

_exit(code)

DESCRIPTION

These functions cause a program to terminate and control to be returned to the operating system.

code is returned to the operating system, as the program's termination code.

exit and *_exit* differ in that *exit* closes all files opened for standard and unbuffered i/o, while *_exit* doesn't.

TECHNICAL INFORMATION

Chapter Contents

Technical Information

This chapter discusses topics that couldn't be conveniently discussed elsewhere.

It's divided into the following sections:

1. *Assembly-language functions*;
2. *C-language interrupt routines.*

1. Assembler Functions

This section discusses assembly-language functions that can be called by, and themselves call, C-language functions. It first discusses the conventions that such functions must follow, and then discusses the in-line placement of assembler statements within C functions.

1.1 Conventions for C-callable, assembly-language functions

A C-callable, assembly-language function must obey the conventions that are described in the following paragraphs.

1.1.1 Names of global variables and functions

By default, the names by which assembly-language modules and C-language modules refer to global variables and functions differ slightly: the assembler name is generated from the C name by prepending an underscore character.

Consider, for example, the following C module:

```
int var;
main()
{
        func(var);
}
```

The names by which an assembler module would by default refer to these global variables and functions are *_var*, *_main*, and *_func*.

You can define an alternate naming convention using the compiler's *+RUx* option. *x* defines how assembler names are derived from C names:

- If *x* is negative, assembly names are derived by prepending an underscore to C names;
- If *x* is zero, assembler names are the same as C names;
- If *x* is positive, assembler names are derived by appending an underscore to C names.

In the following paragraphs, we assume that assembler names are derived from C names by prepending an underscore.

1.1.2 Global variables

A C module's global variables are in either the uninitialized data segment or the initialized data segment.

An assembler module can create an uninitialized variable that can be accessed by a C function, using the *global* directive. For example, the following code creates the global variable *_var*, which can be accessed as an array by a C function, and reserves 8 bytes of storage for it.

```
global __var,8
```

A C function that wants to access *__var* could have the following declaration:

```
extern short var[];
```

To create an initialized variable that can be accessed by a C function, an assembler module can use the *public* and *dc* directives. For example, the following code creates the *public* variable *__ptr* that initially contains a pointer to the symbol *str*, and that can be accessed as a *char* pointer by a C function:

```
        dseg
        public  __ptr
__ptr   dc.l    str
```

To access *__ptr*, a C function could use the following declaration:

```
extern char *ptr;
```

An assembler module can access global initialized or uninitialized variables that are created in C modules by defining the variables with a *public* directive within the *dseg* segment. For example, suppose a C module creates a global, uninitialized *short* named *count* and a global, initialized *short* named *total* using the statement:

```
short count, total=1;
```

An assembler module can access these variables by using the following directives:

```
        dseg
        public  __count, __total
```

1.1.3 Names of external functions and variables

The compiler translates the name of a function or variable to assembly language by truncating the name to 31 characters and optionally adding an underscore to the name (as defined by the *+RUx* option). Thus, to be accessible from C modules or to access C modules, assembler modules must obey this convention.

For example, the following C module calls the function *bmp*, which simply adds 10 to the global *short count*. A C module refers to this function as *bmp*, and an assembler module refers to it as *__bmp*.

```
int count;
main()
{
        bmp();
}
```

An assembler version of *__bmp* could be:

```
        dseg
        public _count
        cseg
        public  _bmp
_bmp:
        add.w   #10,_count
        rts
        end
```

1.1.4 Function calls and returns

The assembler code generated by the compiler for a C call to another function pushes the arguments onto the stack, in the reverse order in which they were specified in the call's argument list, and then calls the function.

An assembler function returns to a C function caller by issuing a *rts* instruction, and leaving the caller's arguments on the stack. The caller then removes the arguments from the stack.

A function returns an integer or pointer in register D0. Floating point values are returned in registers D0 and D1.

For example, consider the following assembler function, *_sub*, that takes two *short* arguments that are passed to it on the stack, subtracts them, and returns the difference as the function value. A C function will refer to this function using the name *sub*.

```
        cseg
        public  _sub
_sub:
        mov     4(sp),d0        ;get first argument
        sub     6(sp),d0        ;subtract second from first
        rts
```

The following C function calls *sub* to subtract *b* from *a*, and stores the difference in *c*:

```
main()
{
        short a,b,c;
        ...
        c = sub(a,b);
}
```

1.1.5 Register usage

An assembler function that is called by a C function must preserve all registers it uses, except for those that the calling function uses for temporary values.

The registers that a module uses for temporary values are defined when the module is compiled, with the *+RT* option; by default, these

are data registers D0-D3 and address registers A0-A2.

1.2 Embedded Assembler Source

Assembler statements can be embedded in a C module by surrounding them with *#asm* and *#endasm* statements. The pound sign (#) must be the first character on the line, and the letters must be lower case.

Embedded assembler code must preserve the contents of all registers it uses, except for those used for temporary values.

It should make no assumptions about the contents of the registers, since the code that the compiler currently generates for C statements may change in the future.

To be safe, a *#asm* statement should be preceded by a semicolon. This avoids problems in which the compiler mistakenly puts a label that is the target of a jump statement after, rather than before, in-line assembly code.

In general, it is safest to contain assembly code in a separate assembler module rather than embedding it in C source.

2. Interrupt Handlers

An interrupt handler can be written in C, with the following provisos: it must have a small assembly language routine that performs the initial and final processing of an interrupt; and it must restrict its use of the library functions. These provisos are discussed in the following paragraphs.

2.1 The Assembly language routine

When the assembly language front-end to a C interrupt handler is activated by an interrupt, it must do the following:

- Save on the stack the registers that the C routine uses for holding temporary values;
- If the C routine uses a small memory model, the assembly language routine must save the small memory model support register and set in it the value *___H1_org+32766.* *___H1_org* is a linker-created symbol whose value is the starting address of the interrupt handler's initialized data segment. In case you can't tell, *___H1_org* begins with two underscores, and has one in the middle;
- *jsr* to the C routine.

It's not necessary for the assembly language routine to save other registers (i.e. registers used for holding the C routine's register variables or the frame pointer register); this will automatically be done by the C routine.

The C routine should return in the usual way; i.e. by executing a *return* instruction or by executing its last instruction. The assembly language routine should then restore all registers that it saved and issue an *rte* instruction.

Here is a sample assembly language routine named *_intbegin.* It saves the default temporary registers D0-D3 and A0-A2; saves and initializes the default small model support register A5; and calls the C language routine whose C name is *intfunc*:

```
        public  _intbegin,_intfunc,___H1_org
_intbegin
        movem.l d0-d3/a0-a2/a5,-(sp)
        move.l  #___H1_org+32766,a5
        jsr     _intfunc
        movem.l (sp)+,d0-d3/a0-a2/a5
        rte
```

2.2 Use of library functions by C-language interrupt routines

A C interrupt routine can call the reentrant library functions that are provided with Aztec C68k/ROM; it usually shouldn't call the non-reentrant library functions. A function is reentrant if it doesn't access

global or static variables, and is non-reentrant if it does.

The non-reentrant library functions are these:

- The high-level buffer-allocation functions *malloc*, *free*, etc.
- *sprintf*;
- *sscanf*;
- The standard i/o functions, usually;
- The unbuffered i/o functions, usually.

The standard i/o functions are not reentrant, because they have global control blocks and because they call the non-reentrant *malloc* and *free* functions. An interrupt routine can call the standard i/o functions if those calls meet certain requirements: the calls can't modify control block fields that may be accessed by the standard i/o calls of an interrupted process, and they can't call *malloc* or *free*. For example, an interrupt routine can issue standard i/o calls to pre-opened streams whose standard i/o operations are unbuffered. It can also issue standard i/o calls to pre-opened buffered streams, if the buffer has been preallocated, and if it only accesses those streams.

The unbuffered i/o functions (which you must write) are usually not reentrant, because they usually have a global table. But an interrupt routine can call the unbuffered i/o functions if those calls don't modify fields that may be accessed by the calls of an interrupted process.