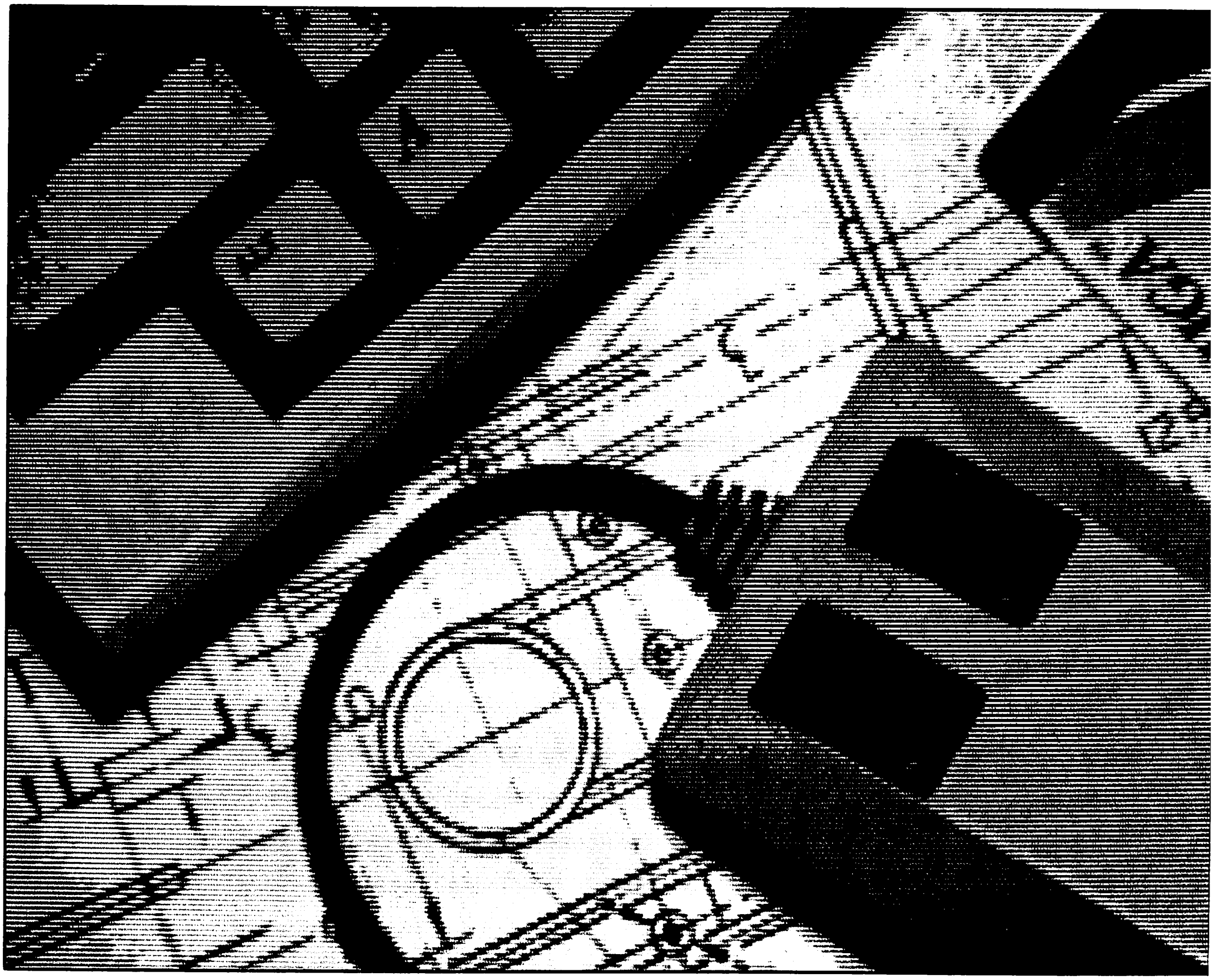
AMIGA T.M.



AmigaDOS Developer's Manual

AmigaDOS Developer's Manual

Acknowledgements

This manual was originally written by Tim King and then completely revised by Jessica King.

A special thanks to Patria Brown whose editorial suggestions substantially contributed to the quality of the manual.

Also thanks to Bruce Barrett, Keith Stobie, Robert Peck and all the others at Commodore-Amiga who carefully checked the contents; to Tim King, Paul Floyd, and Alan Cosslett who did the same at Metacomco; and to Pamela Clare and Liz Laban who spent many hours carefully proof-reading each version.

COPYRIGHT

DISCLAIMER

THIS PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE PROGRAM IS ASSUMED BY YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT THE DEVELOPER OR COMMODORE-AMIGA, INC. OR ITS PARENT AND AFFILIATED DEALERS) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. FURTHER, COMMODORE-AMIGA, INC. OR ITS PARENT AND AFFILIATED COMPANIES DO NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF THE PROGRAM IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE; AND YOU RELY ON THE PROGRAM AND RESULTS SOLELY AT YOUR OWN RISK. IN NO EVENT WILL COMMODORE-AMIGA, INC. OR ITS PARENT COMPANY AND AFFILIATED COMPANIES BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTAL DAMAGES RESULTING FROM ANY DEFECT IN THE PROGRAM EVEN IF IT HAS BEEN ADVISED OF THE POSSIBILITY OF IMPLIED WARRANTIES OR LIABILITIES FOR INCIDENTAL OR CONSEQUENTAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY.

This manual refers to Release 1, August 1985.

Printed in the U.S.A.

CBM Product Number 327265-01 rev 1.0 8.27.85

Using Preferences

If you have a working version of Preferences and you change the text size, for example, from 60 to 80, then AmigaDOS renders any new windows that you create in 80 columns. However, any old windows in the system remain with a text size of 60. To incorporate text size into the system, you need to create a new window, select the old window, and finally delete the old window.

Follow these steps:

1. Use the newcli command.
2. Select the old window.
3. Use the endcli command in the old window to delete the old window.

If you alter the CLI selection, the change may not take effect immediately. If you give the new preferences and re-boot, they take effect.

Table of Contents

1. Programming on the Amiga

2. Calling AmigaDOS

3. The Macro Assembler

4. The Linker

Chapter 1: Programming on the Amiga

This chapter introduces the reader to programming in C or Assembler under AmigaDOS.

Table of Contents

1.1. Introduction

1.2 Program Development for the Amiga

1.2.1 Getting Started

1.2.2 Calling Resident Libraries

1.2.3 Creating an Executable Program

1.3 Running a Program Under the CLI

1.3.1 Initial Environment in Assembler

1.3.2 Initial Environment in C

1.3.3 Failure of Routines

1.3.4 Terminating a Program

1.4 Running a Program Under the Workbench

1.5 Cross Development

1.5.1 Cross Development on a Sun

1.5.2 Cross Development under MSDOS

1.5.3 Cross Development on Other Computers

1.1 Introduction

The AmigaDOS programming environment is available on the Amiga, Sun, and IBM PC.

This manual assumes that you have some familiarity with either C or Assembler. It does not attempt to teach either of these languages. An introduction to C can be found in the book *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice Hall. There are a number of books on writing 68000 assembler, including *Programming the MC68000* by Tim King and Brian Knight, published by Addison Wesley.

1.2 Program Development for the Amiga

This section describes how to develop programs for the Amiga. It describes what you need before you start, how you can call the system routines, and how to create a file that you can execute on the Amiga.

WARNING: Before you do ANYTHING, you should make a backup copy of your system disk. For instructions, see the section, "Backing Up," at the beginning of the *AmigaDOS User's Manual.*

1.2.1 Getting Started

Before you start writing programs for the Amiga, you need the following items:

1. Documentation on AmigaDOS and other system routines that you can call. For example, you need the *AmigaDOS User's Manual, ROM Kernel Manual,* and possibly the *AmigaDOS Technical Reference Manual* as well.

2. Documentation on the language you intend to use. If you intend to use Assembler or C, then this manual tells you how to use these tools although it does not contain any specific information normally found in a language reference manual.

3. Header files containing the necessary Amiga structure definitions and the values for calling the system routines that you need. Commodore-Amiga provides these header files as include files for either C (usually ending in .h) or assembler (ending in .i). To use a particular resident library, you must include one or more header files containing the relevant definitions. For example, to use AmigaDOS from C, you must include the file 'dos.h.'

4. An assembler or compiler either running on the Amiga itself or on one of the cross development environments.

5. The Amiga linker, again running on the Amiga or on another computer, as well as the standard Amiga library containing functions, interface routines, and various absolute values.

6. Tools to download programs if you are using a cross-development environment.

1.2.2 Calling Resident Libraries

You should note that there are two ways of calling system routines from a user assembly program. C programmers simply call the function as specified. You usually call a system routine in assembler by placing the library base pointer for that resident library in register A6 and then jumping to a suitable negative offset from that pointer. The offsets are available to you as absolute externals in the Amiga library, with names of the form LVO_name. So, for instance, a call could be JSR LVO_name(A6), where you have loaded A6 with a suitable library base pointer. These base pointers are available to you from the OpenLibrary call to Exec; you can find the base pointer for Exec at location 4 (the only absolute location used in the Amiga). This location is also known as AbsExecBase which is defined in amiga.lib. (See the *ROM Kernel Manual* for further details on Exec.)

You can call certain RAM-based resident libraries and the AmigaDOS library in this way, if required. Note that the AmigaDOS library is called 'dos.library'. However, you do not need to use A6 to hold a pointer to the library base; you may use any other register if you need to. In addition, you may call AmigaDOS using the resident library call feature of the linker. In this case, simply code a JSR to the entry point and the linker notes the fact that you have used a reference to a resident library. When your code is loaded into memory, the loader automatically opens the library and closes it for you when you have unloaded. The loader automatically patches references to AmigaDOS entry points to refer to the correct offset from the library base pointer.

1.2.3 Creating an Executable Program

To produce a file that you can execute on the Amiga, you should follow the four steps below. You can do each step either on the Amiga itself or on a suitable cross development computer.

1. Get your program source into the Amiga. To do this, you can type it directly in using an editor, or you can transfer it from another computer. Note that you can use the READ and DOWNLOAD programs on the Amiga to transfer character or binary files.

2. Assemble or compile your program.

3. Link your program together, including any startup code you may require at the beginning, and scan the Amiga library and any others you may need to satisfy any external references.

4. Load your program into the Amiga and watch it run!

1.3 Running a Program Under the CLI

There are two ways you can run a program. First, you can run your program under a CLI (Command Line Interface). Second, you can run your program under the Workbench. This section describes the first of the two ways.

Running a program under the CLI is a little like using an old-fashioned line-oriented TTY system although you might find a CLI useful, for example, to port your program over to your Amiga as a first step in development. To load and enter your program, you simply type the name of the file that contains the binary and possibly follow this with a number of arguments.

1.3.1 Initial Environment in Assembler

When you load a program under a CLI, you type the name of the program and a set of arguments. You may also specify input or output redirection by means of the '>' and '<' symbols. The CLI automatically provides all this information for the program when it starts up.

When the CLI starts up a program, it allocates a stack for that program. This stack is initially 4000 bytes, but you may change the stack size with the STACK command. AmigaDOS obtains this stack from the general free memory heap just before you run the program; it is not, however, the same as the stack that the CLI uses. AmigaDOS pushes a suitable return address onto the stack that tells the CLI to regain control and unload your program. Below this on the stack at 4(SP) is the size of the stack in bytes, which may be useful if you wish to perform stack checking.

Your program starts with register A0 pointing to the arguments you, or anyone else running your program typed. AmigaDOS stores the argument line in memory within the CLI stack and this pointer remains valid throughout your program. Register D0 indicates the number of characters in the argument line. You can use these initial values to decode the argument line to find out what the user requires. Note that all registers may be corrupted by a user program.

To make the initial input and output file handles available, you call the AmigaDOS routines Input() and Output(). Remember that you may have to open the AmigaDOS library before you do this. The calls return file handles that refer to the standard input and output the user requires. This standard input and output is usually the terminal unless you redirected the I/O by including '>' or '<' on the argument line. You should not close these file handles with your program; the CLI opened them for you and it will close them, if required.

1.3.2 Initial Environment in C

When programming in C, you should always include the startup code as the first element in the linker input. This means that the linker enters your program at the startup code entry point. This section of code scans the argument list and makes the arguments available in 'argv', with the number of arguments in 'argc' as usual. It also opens the AmigaDOS library and calls Input() and Output() for you, placing the resulting file handles into 'stdin' and 'stdout'. It then calls the C function 'main.'

1.3.3 Failure of Routines

Most AmigaDOS routines return a zero if they fail; the exceptions are the Read and Write calls that return -1 on finding an error. If you receive an error return, you can call IoErr() to obtain more information on the failure. IoErr() returns an integer that corresponds to a full error code, and you may wish to take different actions depending on exactly why the call failed. A complete list of error codes and messages can be found at the end of the *AmigaDOS User's Manual*.

1.3.4 Terminating a Program

To exit from a program, it is sufficient to give a simple RTS using the initial stack pointer (SP). In this case, you should provide a return code in register D0. This is zero if your program succeeded; otherwise, it is a positive number. If you return a non-zero number, then the CLI notices an error. Depending on the current fail value (set by the command FAILAT), a non-interactive CLI, such as one running a command sequence set up by the EXECUTE command, terminates. A program written in C can simply return from 'main' which returns to the startup code; this clears D0 and performs an RTS.

Alternatively a program may call the AmigaDOS function Exit, which takes the return code as argument. This instructs your program to exit no matter what value the stack pointer has.

It is important at this stage to stress that AmigaDOS does not control any resources; this is left entirely up to the programmer. Any files that a user program opens must be closed before the program terminates. Likewise, any locks it obtains must be freed, any code it loads must be unloaded, and any memory it allocates returned. Of course, there may be cases where you do not wish to return all resources, for example, when you have written a program that loads a code segment into memory for later use. This is perfectly acceptable, but you must have a mechanism for eventually returning any memory, file locks, and so on.

1.4 Running a Program under the Workbench

To run a program under the Workbench, you need to appreciate the different ways in which a program may be run on the Amiga. Under the CLI your program is running as part of the CLI process. It can inherit I/O streams and other information from the CLI, such as the arguments you provided.

If a program is running under the Workbench, then AmigaDOS starts it as a new process running at the same time as Workbench. Workbench loads the program and then sends a message to get it started. You must therefore wait for this initial message before you start to do anything. You must retain the message and return it back to Workbench when your program has finished, so that Workbench can unload the code of your program.

For C programmers, this is all done by simply using a different startup routine. For assembly language programmers, this work must be done yourself.

You should also note that a program running as a new process initiated by Workbench has no default input and output streams. You must ensure that your program opens all the I/O channels that it needs, and that it closes them all when it has finished.

1.5 Cross Development

If you are using a cross-development environment, then you need to download your code onto the Amiga. This section describes the special support Commodore-Amiga gives to Sun and MSDOS environments. It also describes how to cross-develop in other environments without this special support.

1.5.1 Cross Development on a Sun

The tools available on the Sun for cross development include the assembler, linker, and two C compilers. The argument formats of the assembler and linker on the Sun are identical to those on the Amiga when running under the CLI. The Greenhills C compiler is only available on the Sun and is described here.

The compiler is called metacc, and it accepts several types of files. It assumes that filenames ending in .c represent C source programs. The compiler then compiles these .c files and places the resulting object program in the current directory with the same filename, but ending with .obj. The suffix .obj denotes an object file. The compiler assumes that files ending in .asm are assembly source programs. You can use the assembler to assemble these and produce an object file (ending with .obj) in the current directory.

The compiler metacc takes many options with the following format:

```
metacc [<opt1>[,<opt2>[,..<optn>]]] [<file>[,...<filen>]]
```

The options available are as follows:

```
-c -g -go -w -p -pg -O[<optflags>] -fsingle
-S -E -C -X70 -o <output> -D <name=def>
-U <name> -I <dir> -B <string> -t [p012]
```

The following options instruct metacc to

-c just compile the program, suppressing the loading phase of the compilation, and forcing an object file to be produced even if it only compiles one program.

-g produce additional symbol table information for the debugger dbx and to pass the -lg flag to ld.

-go produce additional symbol table information in an older format set by the adb debugger. Also, pass the -lg flag to ld.

-w suppress all warning messages.

-p produce profiling code to count the number of times each routine is called. If loading takes place, replace the standard startup routine by one that is automatically called by the monitor and uses a special profiling library instead of the standard C library.

Use the prof program to generate an execution profile.

pg produce profiling code like -p, but invokes a run-time recording mechanism that keeps more extensive statistics and produces a gmon.out file at normal termination.

Use the gprof program to generate an execution profile.

-O [<optflags>] use the object code optimizer to improve the generated code.

If 'optflags' appears, you include <optflags> in the command line to run the optimizer. You can use -O to pass option flags.

-fsingle use single-precision arithmetic in computations involving only float numbers - that is, do not convert everything to double (that is, the default).

Note: Floating-point parameters are still converted to double-precision, and functions that return values still return double-precision values.

WARNING: Certain programs run much faster using the fsingle option, but beware that you can lose significance due to lower precision intermediate values.

-S compile the specified C program(s) and leave the assembler-language output on corresponding files ending with .obj.

-E run only the C preprocessor on the named C program(s) and send the result to the standard output.

-C prevent the C preprocessor from removing comments.

-X70 generate code using Amiga floating point format. This code is compatible with the floating point math ROM library provided on the Amiga.

-o <output> name the final output file 'output'. If you use this option, the file a.out is left undisturbed.

-D<name=def> define 'name' to the preprocessor, as if by #define. If no definition is given, define the name as '1'.

-U<name> remove any initial definition of 'name'.

-I<dir> always look for #include files whose names do not begin with '/' first in the directory of the <file> argument, then look in the <dir> specified in the -I option, and finally look in the /usr/include directory.

-B<string> find substitute compiler passes in the files specified by <string> with the endings cpp, ccom, and c2. If 'string' is empty, use a backup version.

-t[p012] find only the designated compiler passes in the files whose names are constructed by a -B option. In the absence of a -B option, assume <string> to be /usr/new/.

The letter and number combinations that you can specify for the -t option have the following meanings:

p	cpp - the C preprocessor
0	metacom - both phases of the C compiler, but not the optimizer.
1	Ignored in this system - this option would be for the second phase of a two-phase compiler but in the Sun system; ccom includes both phases.
2	c2 - the object code optimizer.

The compiler metacc assumes that other arguments are loaded option arguments, object programs, or libraries of object programs. Unless you specify -c, -S, or -E, metacc loads these programs and libraries together with the results of any compilations or assemblies specified, (in the order given) to produce an executable program named a.out. To override the name a.out, you can use the loader's -o<name> option.

If a single C program is compiled and loaded all at once, the intermediate .o file is deleted.

Figure 1-A lists the filenames of special metacc files and their descriptions.

Special Files

File Description	Filename
C source code	file.c
Assembler source file	file.asm
Object file	file.o
Library of object files	file.lib
Executable output files	a.out
Temporary files	/tmp/ctm
Preprocessor	/lib/cpp
Compiler	/lib/ccom
Optional optimizer	/lib/c2
Runtime startoff	/lib/crt0.o
Startoff for profiling	/lib/mcrt0.o
Startoff for gprof-profiling	/usr/lib/gcrt0.o
Standard library	/lib/libc.a
Profiling library	/usr/lib/libc__p.a
Standard directory (#include.	/usr/include
Files produced for analysis by prof	mon.out
File produced for analysis by gprof	gmon.out

Figure 1.A: Special metacc Filenames

You can download the files you produce from the linker on the Sun to your Amiga in three ways: the first, and by far the easiest, requires a BillBoard; the second requires a parallel port; and the third requires a serial line.

If you have the special hardware device called a BillBoard, you can download your linked load file (by convention this should end with .ld) as follows:

1. Startup the program 'binload' on the Sun

   ```
   binload -p &
   ```

   (this need only be done once)

2. then on the Amiga, type

   ```
   download <sun filename> <amiga filename>
   ```

3. To run the program, type

   ```
   <amiga filename>
   ```

For example:

On the Sun, type

```
binload -p &
```

On the Amiga, type

```
download test.ld test
```

or type

```
download /usr/commodore/amiga/V24/examples/DOS/test.ld test
```

then type

```
test
```

Note that DOWNLOAD gains access to files on the Sun relative to the directory where binload started. If the directory on the Sun was /usr/commodore/amiga/V24/examples/DOS as above, the filename test.ld is all that is necessary. If you cannot remember the directory where binload started, you must specify the full name. To stop binload, do a 'ps' and then a 'kill' on its PID. Note that the soft reset of the computer tells binload to write a message to its standard output (the default is the window where it started). If the transfer hangs, press CTRL-C at the Amiga to kill DOWNLOAD. (See the *AmigaDOS User's Manual* for further information on the AmigaDOS control conventions CTRL-C, CTRL-D, CTRL-E, and CTRL-F.)

If you do not have a BillBoard, you can download files through a parallel port. To do this, follow these steps:

1. Send the download ASCII files to the Amiga via the parallel port by typing

   ```
   send demo.ld
   ```

   If you do not give 'send' any arguments, the standard input is used. The default output device is /dev/lp0, which is usually correct. To change the default output, use the -o argument.

2. On the Amiga, type the following:

   ```
   READ demo
   ```

   READ then reads characters from the parallel port and places them in the file named 'demo'.

3. Once READ has finished, type

   ```
   demo
   ```

   to run the program demo.

You can also download files serially. To do this, follow these steps:

1 Convert the Binary Load File into an ASCII hex file ending with Q by typing

   ```
   convert <demo.ld  >demo.dl
   ```

   (where .dl, by convention, stands for DownLoad). The above rule exists in the included makefile, makeamiga. (See the *AmigaDOS Technical Reference Manual* for further details on the Amiga Binary Load files.)

2. Type

   ```
   tip amiga
   ```

3. On the Amiga, type

   ```
   READ demo serial
   ```

4. Within tip, type

   ```
   ~> demo.dl
   ```

5. When the READ completes on the Amiga, type the filename 'demo' to run it

---

WARNING: The Sun serial link often hangs for no apparent reason. Reboot the Sun if this happens.

If the Sun serial link should happen to hang, reboot the Sun, then type

```
tip
```

and within tip, type

```
Q
```

to get the READ on the Amiga to complete. Once this is done, start a new READ and type the following symbols on the Sun

```
            >
```

1.5.2 Cross Development under MSDOS

To cross develop on a computer running MSDOS for your Amiga, you need various tools that are supplied in the directory \V25\bin. These include the C compiler, assembler, and linker as well as commands to assist in downloading. You use the same syntax for the tools running under MSDOS as under the CLI on the Amiga.

To download via an IBM PC serial port (called AUX), follow these steps:

1. Type on your Amiga:

   ```
   READ file SERIAL
   ```

2. On the PC, type

   ```
   convert <file.ld  >AUX:
   ```

3. On your Amiga, you can now type

   ```
   file
   ```

   to try the program out.

1.5.3 Cross Development on Other Computers

You'll need to have a suitable cross compiler or assembler, and the include files defining all the entry points. You'll also need either the Amiga linker ALINK running on your equipment or on the Amiga. Finally you'll need a way to convert a binary file into a hexadecimal stream terminated with a Q (as this is the way that READ accepts data), and a way of putting this data out from a serial or parallel port.

Once you have created a suitable binary file, you must transfer this to the Amiga using the READ command (as described in Section 1.5.2 of this manual). If you have the Amiga linker running on your computer, then you can transfer complete binary load files; otherwise, you'll have to transfer binary object files in the format accepted by ALINK, and then perform the link step on the Amiga.

Chapter 2: Calling AmigaDOS

This chapter describes the functions provided by the AmigaDOS resident library. To help you, it provides the following: an explanation of the syntax, a full description of each function, and a quick reference card of the available functions.

Table of Contents

2.1 Syntax

2.2 AmigaDOS Functions

Quick Reference Card

2.1 Syntax

The syntax used in this chapter shows the C function call for each AmigaDOS function and the corresponding register you use when you program in assembler.

1. Register values

The letter/number combinations (D0...Dn) represent registers. The text to the left of an equals sign represents the result of a function. A register (that is, D0) appearing under such text indicates the register value of the result. Text to the right of an equals sign represents a function and its arguments, where the text enclosed in parentheses is a list of the arguments. A register (for example, D2) appearing under an argument indicates the register value of that argument.

Note that not all functions return a result.

2. Case

The letter case (that is, lower or upper case) IS significant. For example, you must enter the word 'FileInfoBlock' with the first letter of each component word in upper case.

3. Boolean returns

-1 (TRUE or SUCCESS), 0 (FALSE or FAILURE).

4. Values

All values are longwords (that is, 4 byte values or 32 bits). Values referred to as "string" are 32 bit pointers to NULL terminated series of characters.

5. Format, Argument and Result

Look at 'Argument:' and 'Result:' for further details on the syntax used after 'Format:'. Result describes what is returned by the function (that is, the left of the equal sign). Argument describes what the function expects to work on (that is, the list in parentheses). Figure 2-A should help explain the syntax.

```
Format of function      result = Function( argument )
                        Register           Register

Example                 lock = CreateDir( name )
                        D0                D1
```

Figure 2-A: Format of Functions and Registers

2.2 AmigaDOS Functions

This reference section describes the functions provided by the AmigaDOS resident library. Each function is arranged alphabetically under the following headings: File Handling, Process Handling, and Loading Code. These headings indicate the action of the functions they cover. Under each function name, there is a brief description of the function's purpose, a specification of the format and the register values, a fuller description of the function, and an explanation of the svntax of the arguments and result. To use any of these functions, you must link with amiga.lib.

File Handling

CLOSE

Purpose: To close a file for input or output.

Form:

```
Close( file )
       D1
```

Argument: file - file handle

Description:

The file handle 'file' indicates the file that Close snould close. You obtain this file handle as a result of a call to Open. You must remember to close explicitly all the files you open in a program. However, you should not close inherited file handles opened elsewhere.

CREATEDIR

Purpose: To create a new directory.

Form:

```
lock = CreateDir( name )
D0                D1
```

Argument: name - string

Result: lock - pointer to a lock

Description:

CreateDir creates a new directory with the name you specified, if possible. It returns an error if it fails. Remember that AmigaDOS can only create directories on devices which support them, for example, disks.

A return of zero means that AmigaDOS has found an error (such as: disk write protected), you should then call IoErr(); otherwise, CreateDir returns a shared read lock on the new directory.

CURRENTDIR

Purpose: To make a directory associated with a lock the current working directory.

Form:

```
oldLock = CurrentDir( lock )
D0                    D1
```

Argument: lock - pointer to a lock

Result: oldLock - pointer to a lock

Description:

CurrentDir makes current a directory associated with a lock. (See also LOCK). It returns the old current directory lock.

A value of zero is a valid result here and indicates that the current directory is the root of the initial start-up disk.

DELETEFILE

Purpose: To delete a file or directory.

Form:

```
success = DeleteFile( name )
D0                    D1
```

Argument: name - string

Result: success - boolean

Description:

DeleteFile attempts to delete the file or directory 'name'. It returns an error if the deletion fails. Note that you must delete all the files within a directory before you can delete the directory itself.

DUPLOCK

Purpose: To duplicate a lock.

Form:

```
newLock = DupLock( lock )
D0                 D1
```

Argument: lock - pointer to a lock

Result: newLock - pointer to a lock

Description:

DupLock takes a shared filing system read lock and returns another shared read lock to the same object. It is impossible to create a copy of a write lock. (For more information on locks, see under LOCK.)

EXAMINE

Purpose: To examine a directory or file associated with a lock.

Form:

```
success = Examine( lock, FileInfoBlock )
 D0                D1    D2
```

Argument: lock - pointer to a lock
FileInfoBlock - pointer to a file info block

Result: success - boolean

Description:

Examine fills in information in the FileInfoBlock concerning the file or directory associated with the lock. This information includes the name, size, creation date, and whether it is a file or directory.

Note: FileInfoBlock must be longword aligned. You can ensure this in the language C if you use Allocmem. (See the *ROM Kernel Manual* for further details on the exec call Allocmem.)

Examine gives a return code of zero if it fails.

EXNEXT

Purpose: To examine the next entry in a directory.

Form:

```
success = ExNext( lock, FileInfoBlock )
 D0                 D1     D2
```

Argument: lock - pointer to a lock
FileInfoBlock - pointer to a file info block

Result: success - boolean

Description:

This routine is passed a lock, usually associated with a directory, and a FileInfoBlock filled in by a previous call to Examine. The FileInfoBlock contains information concerning the first file or directory stored in the directory associated with the lock. ExNext also modifies the FileInfoBlock so that subsequent calls return information about each following entry in the directory.

ExNext gives a return code of zero if it fails for some reason. One reason for failure is reaching the last entry in the directory. However, IoErr() holds a code that may give more information on the exact cause of a failure. When ExNext finishes after the last entry, it returns ERROR_NO_MORE_ENTRIES

So, follow these steps to examine a directory:

1) Use Examine to get a FileInfoBlock about the directory you wish to examine.

2) Pass ExNext the lock related to the directory and the FileInfoBlock filled in by the previous call to Examine.

3) Keep calling ExNext until it fails with the error code held in IoErr() equal to ERROR_NO_MORE_ENTRIES.

4) Note that if you don't know what you are examining, inspect the type field of the FileInfoBlock returned from Examine to find out whether it is a file or a directory which is worth calling ExNext for.

The type field in the FileInfoBlock has two values: if it is negative, then the file system object is a file; if it is positive, then it is a directory.

INFO

Purpose: Returns information about the disk.

Form:

```
success = Info( lock, Info_Data )
 D0               D1     D2
```

Argument: lock - pointer to a lock
Info__Data - pointer to an Info__Data structure

Result: success - boolean

Description:

Info finds out information about any disk in use. 'lock' refers to the disk, or any file on the disk. Info returns the Info__Data structure with information about the size of the disk, number of free blocks and any soft errors. Note that Info__Data must be longword aligned.

INPUT

Form:

```
file = Input()
 D0
```

Result: file - file handle

Description:

To identify the program's initial input file handle, you use Input. (To identify the initial output, see under OUTPUT.)

IOERR

Purpose: To return extra information from the system.

Form:

```
error = IoErr()
 D0
```

Result: error - integer

Description:

I/O routines return zero to indicate an error. When an error occurs, call this routine to find out more information. Some routines use IoErr(), for example, DeviceProc, to pass back a secondary result.

ISINTERACTIVE

Purpose: To discover whether a file is connected to a virtual terminal or not.

Form:

```
bool = IsInteractive ( file )
 D0                     D1
```

Argument: file - file handle

Result: bool - boolean

Description:

The function IsInteractive gives a boolean return. This indicates whether or not the file associated with the file handle 'file' is connected to a virtual terminal.

LOCK

Purpose: To lock a directory or file.

Form:

```
lock  = Lock( name, accessMode )
 D0             D1     D2
```

Argument: name - string
accessMode - integer

Result: lock - pointer to a lock

Description:

Lock returns, if possible, a filing system lock on the file or directory 'name'. If the accessMode is ACCESS_READ, the lock is a shared read lock; if the accessMode is ACCESS_WRITE, then it is an exclusive write lock. If LOCK fails (that is, if it cannot obtain a filing system lock on the file or directory) it returns a zero.

Note that the overhead for doing a Lock is less than that for doing an Open, so that, if you want to test to see if a file exists, you should use Lock. Of course, once you've found that it exists, you have to use Open to open it.

OPEN

Purpose: To open a file for input or output.

Form:

```
file = Open( name, accessMode )
 D0            D1     D2
```

Argument: name - string accessMode - integer

Result: file - file handle

Description:

Open opens 'name' and returns a file handle. If the accessMode is MODE_OLDFILE (=1005), OPEN opens an existing file for reading or writing. However, Open creates a new file for writing if the value is MODE_NEWFILE (=1006). The 'name' can be a filename (optionally prefaced by a device name), a simple device such as NIL:, a window specification such as CON: or RAW: followed by window parameters, or *, representing the current window.

For further details on the devices NIL:, CON:, and RAW:, see Chapter 1 of the of the *AmigaDOS User's Manual*. If Open cannot open the file 'name' for some reason, it returns the value zero (0). In this case, a call to the routine IoErr() supplies a secondary error code.

For testing to see if a file exists, see the entry under LOCK.

OUTPUT

Form:

```
          file = Output()
            D0
```

Result: file - file handle

Description:

To identify the program's initial output file handle, you use Output. (To identify the initial input, see under INPUT.)

PARENTDIR

Purpose: To obtain the parent of a directory or file.

Form:

```
Lock = ParentDir( lock )
 D0               D1
```

Argument: lock - pointer to a lock

Result: lock - pointer to a lock

Description:

This function returns a lock associated with the parent directory of a file or directory. That is, ParentDir takes a lock associated with a file or directory and returns the lock of its parent directory.

Note: The result of ParentDir may be zero (0) for the root of the current f iling system.

READ

Purpose: To read bytes of data from a file.

Form:

```
actualLength = Read( file, buffer, length )
 D0                  D1    D2      D3
```

Argument: file - file handle
buffer - pointer to buffer
length - integer

Result: actualLength - integer

Description:

You can copy data with a combination of Read and Write. Read reads bytes of information from an opened file (represented here by the argument 'file') into the memory buffer indicated. Read attempts to read as many bytes as fit into the buffer as indicated by the value of length. You should always make sure that the value you give as the length really does represent the size of the buffer. Read may return a result indicating that it read less bytes than you requested, for example, when reading a line of data that you typed at the terminal.

The value returned is the length of the information actually read. That is to say, when 'actualLength' is greater than zero, the value of 'actualLength' is the the number of characters read. A value of zero means that end-of-file has been reached. Errors are indicated by a value of -1. Read from the console returns a value when a return is found or the buffer is full.

A call to Read also modifies or changes the value of IoErr(). IoErr() gives more information about an error (for example, actualLength equals -1) when it is called.

RENAME

Purpose: To rename a directory or file.

Form:

```
success = Rename( oldName, newName )
 D0                 D1       D2
```

Argument: oldName - string
newName - string

Result: success - boolean

Description:

Rename attempts to rename the file or directory specified as 'oldName' with the name 'newName'. If the file or directory 'newName' exists, Rename fails and Rename returns an error.

Both the 'oldName' and the 'newName' can be complex filenames containing a directory specification. In this case, the file will be moved from one directory to another. However, the destination directory must exist before you do this.

Note: It is impossible to rename a file from one volume to another.

SEEK

Purpose: To move to a logical position in a file.

Form:

```
oldPosition = Seek( file, position, mode )
 D0                 D1    D2        D3
```

Argument: file - file handle
position - integer
mode - integer

Result: oldPosition - integer

Description:

Seek sets the read/write cursor for the file 'file' to the position 'position'. Both Read and Write use this position as a place to start reading or writing. If all goes well, the result is the previous position in the file. If an error occurs, the result is -1. You can then use IoErr() to find out more information about the error.

'mode' can be OFFSET_BEGINNING (=-1), OFFSET_CURRENT (=0) or OFFSET_END (=1). You use it to specify the relative start position. For example, 20 from current is a position twenty bytes forward from current, -20 from end is 20 bytes before the end of the current file.

To find out the current file position without altering it, you call to Seek specifying an offset of zero from the current position.

To move to the end of a file, Seek to end-of-file offset with zero position. Note that you can append information to a file by moving to the end of a file with Seek and then writing. You cannot Seek beyond the end of a file.

SETCOMMENT

Purpose: To set a comment.

Form:

```
Success = SetComment( name, comment )
 D0                   D1    D2
```

Argument: name - file name
comment - pointer to a string

Result: success - boolean

Description:

SetComment sets a comment on a file or directory. The comment is a pointer to a null-terminated string of up to 80 characters.

SETPROTECTION

Purpose: To set file, or directory, protection.

Form:

```
Success = SetProtection( name, mask )
 D0                        D1    D2
```

Argument: name - file name
mask - the protection mask required

Result: success - boolean

Description:

SetProtection sets the protection attributes on a file or directory. The lower four bits of the mask are as follows:

bit 3: if 1 then reads not allowed, else reads allowed.
bit 2: if 1 then writes not allowed, else writes allowed.
bit 1: if 1 then execution not allowed, else execution allowed.
bit 0: if 1 then deletion not allowed, else deletion allowed.

Bits 31-4 Reserved.

Only delete is checked for in the current release of AmigaDOS. Rather than referring to bits by number you should use the definitions in "include/libraries/dos.h".

UNLOCK

Purpose: To unlock a directory or file.

Form:

```
UnLock( lock )
        D1
```

Argument: lock - pointer to a lock

Description:

UnLock removes a filing system lock obtained from Lock, DupLock, or CreateDir.

WAITFORCHAR

Purpose: To indicate whether characters arrive within a time limit or not.

Form:

```
bool = WaitForChar( file, timeout )
 D0                  D1    D2
```

Argument: file - file handle
timeout - integer

Result: bool - boolean

Description:

If a character is available to be read from the file associated with the handle 'file' within a certain time, indicated by 'timeout', WaitForChar returns -1 (TRUE); otherwise, it returns 0 (FALSE). If a character is available, you can use Read to read it. Note that WaitForChar is only valid when the I/O streams are connected to a virtual terminal device. 'timeout' is specified in microseconds.

WRITE

Purpose: To write bytes of data to a file.

Form:

```
returnedLength =  Write( file, buffer, length )
 D0                      D1    D2      D3
```

Argument: file - file handle
buffer - pointer to buffer
length - integer

Result: returnedLength - integer

Description:

You can copy data with a combination of Read and Write. Write writes bytes of data to the opened file 'file'. 'length' refers to the actual length of data to be transferred; 'buffer' refers to the buffer size.

Write returns a value that indicates the length of information actually written. That is to say, when 'length' is greater than zero, the value of 'length' is the number of characters written. A value of -1 indicates an error. The user of this call must always check for an error return which may, for example, indicate that the disk is full.

Process Handling

CREATEPROC

Purpose: To create a new process.

Form:

```
process = CreateProc( name, pri, segment, stackSize )
  D0                  D1    D2    D3       D4
```

Argument:

name - string
pri - integer
segment - pointer to a segment
stackSize - integer

Result: process - process identifier

Description:

CreateProc creates a process with the name 'name'. That is to say, CreateProc allocates a process control structure from the free memory area and then initializes it.

CreateProc takes a segment list as the argument 'segment'. (See also under LOADSEG and UNLOADSEG.) This segment list represents the section of code that you intend to run as a new process. CreateProc enters the code at the first segment in the segment list, which should contain suitable initialization code or a jump to such.

'stackSize' represents the size of the root stack in bytes when CreateProc activates the process. 'pri' specifies the required priority of the new process. The result is the process identifier of the new process, or zero if the routine failed.

The argument 'name' specifies the process name.

A zero return code implies an error of some kind.

DATESTAMP

Purpose: To obtain the date and time in internal format.

Form:

```
v:= DateStamp( v )
```

Argument: v - pointer

Description:

DateStamp takes a vector of three longwords that is set to the current time. The first element in the vector is a count of the number of days. The second element is the number of minutes elapsed in the day. The third is the number of ticks elapsed in the current minute. A tick happens 50 times a second. DateStamp ensures that the day and minute are consistent. All three elements are zero if the date is unset. DateStamp currently only returns even multiples of 50 ticks. Therefore the time you get is always an even number of ticks.

DELAY

Purpose: To delay a process for a specified time.

Form:

```
Delay( timeout )
          D1
```

Argument: timeout - integer

Description:

The function Delay takes an argument 'timeout'. 'timeout' allows you to specify how long the process should wait in ticks (50 per second).

DEVICEPROC

Purpose: To return the process identifier of the process handling that I/O.

Form:

```
process = DeviceProc( name )
 D0                     D1
```

Argument: name - string

Result: process - process identifier

Description:

DeviceProc returns the process identifier of the process that handles the device associated with the specified name. If DeviceProc cannot find a process handler, the result is zero. If 'name' refers to a file on a mounted device, then IoErr() returns a pointer to a directory lock.

You can use this function to determine the process identification of the handler process where the system should send its messages.

EXIT

Purpose: To exit from a program.

Form:

```
Exit( returnCode )
        D1
```

Argument: returnCode - integer

Description:

Exit acts differently depending on whether you are running a program under a CLI or not. If you run, as a command under a CLI, a program that calls Exit, the command finishes and control reverts to the CLI. Exit then interprets the argument 'returnCode' as the return code from the program.

If you run the program as a distinct process, Exit deletes the process and releases the space associated with the stack, segment list, and process structure.

Loading Code

EXECUTE

Purpose: To execute a CLI command.

Form:

```
Success = Execute( commandString, input, output )
 D0                     D1           D2      D3
```

Argument: commandString - string
input - file handle
output - file handle

Result: Success - boolean

Description:

This function takes a string (commandString) that specifies a CLI command and arguments, and attempts to execute it. The CLI string can contain any valid input that you could type directly at a CLI, including input and output indirection using > and <.

The input file handle will normally be zero, and in this case the EXECUTE command will perform whatever was requested in the commandString and then return. If the input file handle is nonzero then after the (possibly null) commandString is performed subsequent input is read from the specified input file handle until end of file is reached.

In most cases the output file handle must be provided, and will be used by the CLI commands as their output stream unless redirection was specified. If the output file handle is set to zero then the current window, normally specified as *, is used. Note that programs running under the Workbench do not normally have a current window.

The Execute function may also be used to create a new interactive CLI process just like those created with the NEWCLI function. In order to do this you should call Execute with an empty commandString, and pass a file handle relating to a new window as the input file handle. The output file handle should be set to zero. The CLI will read commands from the new window, and will use the same window for output. This new CLI window can only be terminated by using the ENDCLI command. For this command to work the program C:RUN must be present in C:.

LOADSEG

Purpose: To load a load module into memory.

Form:

```
segment = LoadSeg( name )
 D0                 D1
```

Argument: name - string

Result: segment - pointer to a segment

Description:

The file 'name' is a load module produced by the linker. LoadSeg takes this and scatter loads the code segments into memory, chaining the segments together on their first words. It recognizes a zero as indicating the end of the chain.

If an error occurs, Loadseg unloads any loaded blocks and returns a false (zero) result.

If all goes well (that is, LoadSeg has loaded the module correctly), then Loadseg returns a pointer to the beginning of the list of blocks. Once you have finished with the loaded code, you can unload it with a call to UnLoadSeg. (For using the loaded code, see under CREATEPROC.)

UNLOADSEG

Purpose: To unload a segment previously loaded by LOADSEG.

Form:

```
UnLoadSeg( segment )
           D1
```

Argument: segment - pointer to a segment

Description:

UnLoadSeg unloads the segment identifier that was returned by LoadSeg. 'segment' may be zero.

Quick Reference Card

File Handling

Close	to close a file for input or output.
CreateDir	to create a new directory.
CurrentDir	to make a directory associated with a lock the current working directory.
DeleteFile	to delete a file or directory.
DupLock	to duplicate a lock.
Examine	to examine a directory or file associated with a lock.
ExNext	to examine the next entry in a directory.
Info	to return information about the disk.
Input	to identify the initial input file handle.
IoErr	to return extra information from the system.
IsInteractive	to discover whether a file is connected to a virtual terminal or not.
Lock	to lock a file or directory.
Open	to open a file for input or output.
Output	to identify the initial output file handle.
ParentDir	to obtain the parent of a directory or file.
Read	to read bytes of data from a file.
Rename	to rename a file or directory.
Seek	to move to a logical position in a file.
SetComment	to set a comment.
SetProtection	to set file, or directory, protection.
Unlock	to unlock a file or directory.
WaitForChar	to indicate whether characters arrive within a time limit or not.
Write	to write bytes of data to a file.

Process Handling

CreateProc	to create a new process.
DateStamp	to obtain the date and time in internal format.
Delay	to delay a process for a specified time.
DeviceProc	to return the process identifier of the process handling that I/O.
Exit	to exit from a program.

Loading Code

Execute	to execute a CLI command.
LoadSeg	to load a load module into memory
UnloadSeg	to unload a segment previously loaded by LOADSEG.

Chapter 3: The Macro Assembler

This chapter describes the AmigaDOS Macro Assembler. It gives a brief introduction to the 68000 microchip. This chapter is intended for the reader who is acquainted with an assembly language on another computer.

Table of Contents

3.1 Introduction to the 68000 Microchip

3.2 Calling the Assembler

3.3 Program Encoding

3.3.1 Comments

3.3.2 Executable Instructions
3.3.2.1 Label Field
3.3.2.2 Local Labels
3.3.2.3 Opcode Field
3.3.2.4 Operand Field
3.3.2.5 Comment Field

3.4 Expressions

3.4.1 Operators

3.4.2 Operand Types for Operators

3.4.3 Symbols

3.4.4 Numbers

3.5 Addressing Modes

3.6 Variants on Instruction Types

3.7 Directives

3.1 Introduction to the 68000 Microchip

This section gives a brief introduction to the 68000 microchip. It should help you to understand the concepts introduced later in the chapter. It assumes that you have already had experience with assembly language on another computer.

The memory available to the 68000 consists of

- the internal registers (on the chip), and
- the external main memory.

There are 17 registers, but only 16 are available at any given moment. Eight of them are **data registers** named D0 to D7, and the others are **address registers** called A0 to A7. Each register contains 32 bits. In many contexts, you may use either kind of register, but others demand a specific kind. For instance, you may use any register for operations on **word** (16-bit) and **long word** (32-bit) quantities or for indexed addressing of main memory. Although, for operations on **byte** (8-bit) operands, you may only use data registers, and for addressing main memory, you may only use address registers as stack pointers or base registers. Register A7 is the stack pointer, this is in fact two distinct registers; the system stack pointer available in supervisor mode and the user stack pointer available in user mode.

The main memory consists of a number of bytes of memory. Each byte has an identifying number called its **address**. Memory is usually (but not always) arranged so that its bytes have addresses 0, 1, 2, ..., N-2, N-1 where there are N bytes of memory in total. The size of memory that you can directly access is very large - up to 16 million bytes. The 68000 can perform operations on bytes, words, or long words of memory. A word is two consecutive bytes. In a word, the first byte has an even address. A long word is four consecutive bytes also starting at an even address. The address of a long word is the even address of its lowest numbered first byte.

As well as holding items of data being manipulated by the computer, the main memory also holds the **instructions** that tell the computer what to do. Each instruction occupies from one to 5 words, consisting of an **operation word** between zero and four operand words. The operation word specifies what action is to be performed (and implicitly how many words there are in the whole instruction). The **operand words** indicate where in the registers or main memory are the items to be manipulated, and where the result should be placed.

The assembler usually executes instructions one at a time in the order that they occur in memory, like the way you follow the steps in a recipe or play the notes in a piece of written music. There is a special register called the **program counter** (PC) which you use to hold the address of the instruction you want the assembler to execute next. Some instructions, called **jumps** or **branches**, upset the usual order, and force the assembler to continue executing the instruction at a specific address. This lets the computer perform an action repeatedly, or do different things depending on the values of data items.

To remember particular things about the state of the computer, you can use one other special register called the **status register** (SR).

3.2 Calling the Assembler

The command template for assem is

"PROG=FROM/A,-O/K,-V/K,-L/K,-H/K,-C/K,-I/K"

Alternatively, the format of the command line can be described as

```
assem <source file>    [-o <object file>]
                       [-l <listing file>]
                       [-v <verification file>]
                       [-h <header file>]
                       [-c <options>]
                       [-i <include dirlist>]
```

The assembler does not produce an object file or a listing file unless you request them explicitly.

As the assembler is running, it generates diagnostic messages (errors, warnings, and assembly statistics) and sends them to the screen unless you specify a verification file.

To force the inclusion of the named file in the assembly at the head of the source file, you use -h <filename> on the command line. This has the same effect as using

```
INCLUDE "<filename>"
```

on line 1 of the source file.

To set up the list of directories that the assembler should search for any INCLUDEd files, you use the -i keyword. You should specify as many directories as you require after the -i, separating the directory names by a comma (,), a plus sign (+), or a space. Note that if you use a space, you must enclose the entire directory list in double quotes ("). Unix users, however, must escape any double quotes with a backslash (\").

The order of the list determines the order of the directories where the assembler should search for INCLUDEd files. The assembler initially searches the current directory before any others. Thus any file that you INCLUDE in a program must be in the current directory, or in one of the directories listed in the -i list. For instance, if the program 'fred' INCLUDEs, apart from files in the current directory, a file from the directory 'intrnl/incl', a file from the directory 'include/asm', and a file from the directory 'extrnl/incl', you can give the -i directory list in these three ways:

```
assem fred -i intrnl/incl,include/asm,extrnl/incl
assem fred -i intrnl/incl+include/asm+extrnl/incl
assem fred -i "intrnl/incl include/asm extrnl/incl"
```

or, by using the space separator on the Sun under Unix, like this

```
assem fred -i \"intrnl/incl include/asm extrnl/incl\"
```

The -c keyword allows you to pass certain options to the assembler. Each option consists of a single character (in either upper or lower case), possibly followed immediately by a number. Valid options follow here:

S	produces a symbol dump as a part of the object file.
D	inhibits the dumping of local labels as part of a symbol dump. (For C programmers, any label beginning with a period is considered as a local label).
C	ignores the distinction between upper and lower case in labels.
X	produces a cross-reference table at the end of the listing file.

Examples

```
assem fred.asm -o fred.o
```

assembles the file fred.asm and produces an object module in the file fred.o.

```
assem fred.asm -o fred.o -l fred.lst
```

assembles the file fred.asm, produces an object module in the file fred.o, and produces a listing file in fred.lst.

3.3 Program Encoding

A program acceptable to the assembler takes the form of a series of input lines that can include any of the following:

- Comment or Blank lines
- Executable Instructions
- Assembler Directives

3.3.1 Comments

To introduce comments into the program, you can use three different methods:

1. Type a semicolon (;) anywhere on a line and follow it with the text of the comment. For example,

   ```
   CMPA.L A1,A2  ; Are the pointers equal?
   ```

2. Type an asterisk (*) in column one of a line and follow it with the text of the comment. For example,

   ```
   * This entire line is a comment
   ```

3. Follow any complete instruction or directive with a least one space and some text. For example,

   ```
       MOVEQ #IO,DO place initial value in DO
   ```

In addition, note that all blank lines are treated by the assembler as comment lines.

3.3.2 Executable Instructions

The source statements have the general overall format:

[<label>] <opcode> [<operand>[,<operand>]...][<comment>]

To separate each field from the next, press the SPACEBAR or TAB key. This produces a separator character. You may use more than one space to separate fields.

3.3.2.1 Label Field

A label is a user symbol, or programmer-defined name, that either

a) Starts in the first column and is separated from the next field by at least one space, *or*
b) Starts in any column, and is followed immediately with a colon (:).

If a label is present, then it must be the first non-blank item on the line. The assembler assigns the value and type of the program counter, that is, the memory address of the first byte of the instruction or data being referenced, to the label. Labels are allowed on all instructions, and on some directives, or they may stand alone on a line. See the specifications of individual directives in Section 3.7 for whether a label field is allowed.

Note: You must not give multiple definitions to labels. Also, you must not use instruction names, directives, or register names as labels.

3.3.2.2 Local Labels

Local labels are provided as an extension to the MOTOROLA specification. They take the form nnn$ and are only valid between any proper (named) labels. Thus, in this example code segment

```
Labels      Opcodes     Operands

FOO:        MOVE.L      D6,D0
1$:         MOVE.B      (A0)+,(A1)+
            DBRA        D0,1$
            MOVEQ       #20,D0
BAA:        TRAP        #4
```

the label 1$ is only available from the line following the one labelled FOO to the line before the one labelled BAA. In this case, you could then use the label 1$ in a different scope elsewhere in the program.

3.3.2.3 Opcode Field

The Opcode field follows the Label field and is separated from it by at least one space. Entries in this field are of three types.

1. The MC68000 operation codes, as defined in the `MC68000 User Manual`.
2. Assembler Directives.
3. Macro invocations.

To enter instructions and directives that can operate on more than one data size, you use an optional Size-Specifier subfield, which is separated from the opcode by the period (.) character. Possible size specifiers are as follows:

B -	Byte-sized data (8 bits)
W -	Word-sized data (16 bits)
L -	Long Word-sized data (32 bits) or Long Branch specifier
S -	Short Branch specifier

The size specifier must match with the instruction or directive type that you use.

3.3.2.4 Operand Field

If present, the operand field contains one or more operands to the instruction or directive, and must be separated from it by at least one space. When you have two or more operands in the field, you must separate them with a comma (,). The operand field terminates with a space or newline character (a newline character is what the assembler receives when you press RETURN), so you must not use spaces between operands.

3.3.2.5 Comment Field

Anything after the terminating space of the operand field is ignored. So the assembler treats any characters you insert after a space as a comment.

3.4 Expressions

An expression is a combination of symbols, constants, algebraic operators, and parentheses that you can use to specify the operand field to instructions or directives. You may include relative symbols in expressions, but they can only be operated on by a subset of the operators.

3.4.1 Operators

The available operators are listed below in order of precedence.

1. Monadic Minus, Logical NOT (- and ~)
2. Lshift, Rshift (<< and >>)
3. Logical AND, Logical OR (& and !)
4. Multiply, Divide (* and /)
5. Add, Subtract (+ and -)

To override the precedence of the operators, enclose sub-expressions in parentheses. The assembler evaluates operators of equal precedence from left to right. Note that, normally, you should not have any spaces in an expression, as a space is regarded as a delimiter between one field and another.

3.4.2 Operand Types for Operators

In the following table, 'A' represents absolute symbols, and R represents relative symbols. The table shows all the possible operator/operand combinations, with the type of the resulting value. 'x' indicates an error. The Monadic minus and the Logical not operators are only valid with an absolute operand.

Operators	Operands			
	A op A	R op R	A op R	R op A
+	A	x	R	R
-	A	A	x	R
*	A	x	x	x
/	A	x	x	x
&	A	x	x	x
!	A	x	x	x
>>	A	x	x	x
<<	A	x	x	x

Table 3-A: Operand Types for Operators

3.4.3 Symbols

A **symbol** is a string of up to 30 characters. The first character of a symbol must be one of following:

- An alphabetic charcter, that is, a through z, or A through Z.
- An underscore (_).
- A period (.).

The rest of the characters in the string can be any of these characters or also numeric (0 through 9). In all symbols, the lower case characters (a-z) are not treated as synonyms with their upper case equivalents (unless you use the option C when you invoke the assembler). So 'fred' is different from 'FRED' and 'FRed'. However, the assembler recognizes instruction optcodes, directives, and register names in either upper or lower case. A label equated to a register name with EQUR is also recognized by the assembler in either upper or lower case. Symbols can be up to 30 characters in length, all of which are significant. The assembler takes symbols longer than this and truncates them to 30 characters, giving a warning that it has done so. The Instruction names, Directive names, Register names, and special symbols CCR, SR, SP and USP cannot be used as user symbols. A symbol can be one of three types:

Absolute

a) The symbol was SET or EQUated to an Absolute value

Relative

a) The symbol was SET or EQUated to a Relative value

b) The symbol was used as a label

Register

a) The symbol was set to a register name using EQUR (This is an extension from the MOTOROLA specification).

There is a special symbol *, which has the value and type of the current program counter, that is, the address of the current instruction or directive that the assembler is acting on.

3.4.4 Numbers

You may use a number as a term of an expression, or as a single value. Numbers ALWAYS have absolute values and can take one of the following formats:

Decimal
(a string of decimal digits)

Example: 1234

Hexadecimal
($ followed by a string of hex digits)

Example: $89AB

Octal
(@ followed by a string of octal digits)

Example: @743

Binary
(% followed by zeros and ones)

Example: %10110111

ASCII Literal
(Up to 4 ASCII characters within quotes)

Examples: 'ABCD' '*'

Strings of less than 4 characters are justified to the right, using NUL as the packing character.

To obtain a quote character in the string, you must use two quotes. An example of this is

```
'It''s'
```

3.5 Addressing Modes

The effective address modes define the operands to instructions and directives, and you can find a detailed description of them in any good reference book on the 68000. Addresses refer to individual bytes, but instructions, Word and Long Word references, access more than one byte, and the address for these must be word aligned.

In the following table, Dn represents one of the data registers (D0-D7), 'An' represents one of the address registers (A0-A7, SP and PC), 'a' represents an absolute expression, 'r' represents a relative expression, and 'Xn' represents An or Dn, with an optional '.W' or '.L' size specifier. The syntax for each of the modes is as follows:

Table 3-B: Macro Assembler Address Modes and Registers

Address Mode	Description and Examples
Dn	Data Register Direct Example: MOVE D0,D1
An	Address Register Direct Example: MOVEA A0,A1
(An)	Address Register Indirect Example: MOVE D0,(A1)
(An)+	Address Register Indirect Post Increment Example: MOVE (A7)+,D0
-(An)	Address Register Indirect Pre Decrement Example: MOVE D0,-(A7)
a(An)	Address Register Indirect with Displacement Example: MOVE 20(A0),D1
a(An,Xn)	Address Register Indirect with Index Example: MOVE 0(A0,D0),D1 MOVE 12(A1,A0.L),D2 MOVE 120(A0,D6.W),D4

(continuation of 3-B)

Address Mode	Description and Examples
a	Short absolute (16 bits) Example: MOVE $1000,D0
a	Long absolute (32 bits) Example: MOVE $10000,D0
r	Program Counter Relative with Displacement Example: MOVE ABC,D0 (ABC is relative)
r(Xn)	Program Counter Relative with Index Example: MOVE ABC(D0.L),D1 (ABC is relative)
#a	Immediate data Example: MOVE #1234,D0
USP) CCR) SR)	Special addressing modes Example: MOVE A0,USP MOVE D0,CCR MOVE D1,SR

3.6 Variants on Instruction Types

Certain instructions (for example, ADD, CMP) have an **address variant** (that refers to address registers as destinations), **immediate** and **quick** forms (when immediate data possibly within a restricted size range appears as an operand), and a **memory variant** (where both operands must be a postincrement address).

To force a particular variant to be used, you may append A, Q, I or M to the instruction mnemonic. In this case, the assembler uses the specified form of the instruction, if it exists, or gives an error message.

If, however, you specify no particular variant, the assembler automatically converts to the 'I', 'A' or 'M' forms where appropriate. However, it does not convert to the 'Q' form. For example, the assembler converts the following:

```
      ADD.L   A2,A1
```

to

```
      ADDA.L  A2,A1
```

3.7 Directives

All assembler directives (with the exception of DC and DCB) are instructions to the assembler, rather than instructions to be translated into object code. At the beginning of this section, there is a list of all the directives (Table 3-C), arranged by function; at the end there is an individual decription for each directive, arranged by function.

Note that the assembler only allows labels on directives where specified. For example, EQU is allowed a label. It is optional for RORG, but not allowed for LLEN or TTL.

The following table lists the directives by function:

Table 3-C: Directives

Assembly Control

Directive	Description
SECTION	Program section
RORG	Relocatable origin
OFFSET	Define offsets
END	Program end

Symbol Definition

Directive	Description
EQU	Assign permanent value
EQUR	Assign permanent register value
REG	Assign permanent value
SET	Assign temporary value

Data Definition

Directive	Description
DC	Define constants
DCB	Define Constant Block
DS	Define storage

(continuation of 3-C)

Listing Control

Directive	Description
PAGE	Page-throw to listing
LIST	Turn on listing
NOLIST (NOL)	Turn off listing
SPC n	Skip n blank lines
NOPAGE	Turn off paging
LLEN n	Set line length (60 < = n < = 132)
PLEN n	Set page length (24 < = n < = 100)
TTL	Set program title (max 80 chars)
NOOBJ	Disable object code output
FAIL	Generate an assembly error
FORMAT	No action
NOFORMAT	No action

Conditional Assembly

Directive	Description
CNOP	Conditional NOP for alignment
IFEQ	Assemble if expression is 0
IFNE	Assemble if expression is not 0
IFGT	Assemble if expression > 0
IFGE	Assemble if expression > = 0
IFLT	Assemble if expression < 0
IFLE	Assemble if expression < = 0
IFC	Assemble if strings are identical
IFNC	Assemble if strings are not identical
IFD	Assemble if symbol is defined
IFND	Assemble if symbols is not defined
ENDC	End of conditional assembly

Macro Directives

Directive	Description
MACRO	Define a macro name
NARG	Special symbol
ENDM	End of macro definition
MEXIT	Exit the macro expansion

External Symbols

Directive	Description
XDEF	Define external name
XREF	Reference external name

(continuation of 3-C)

General Directives

Directive	**Description**
INCLUDE	Insert file in the source
MASK2	No action
IDNT	Name program unit

Assembly Control Directives

SECTION Program Section

Format: [<label>] SECTION <name>[,<type>]

This directive tells the assembler to restore the counter to the last location allocated in the named section (or to zero if used for the first time).

<name> is a character string optionally enclosed in double quotes.
<type> if included, must be one of the following keywords:

CODE	indicates that the section contains relocatable code. This is the default.
DATA	indicates that the section contains initialized data (only).
BSS	indicates that the section contains uninitialized data

The assembler can maintain up to 255 sections. Initially, the assembler begins with an unnamed CODE section. The assembler assigns the optional symbol <labels> to the value of the program counter after it has executed the SECTION directive. In addition, where a section is unnamed, the shorthand for that section is the keyword CODE.

RORG Set Relative Origin

Format: [<label>] RORG <absexp>

The RORG directive changes the program counter to be <absexp> bytes from the start of the current relocatable section. The assembler assigns relocatable memory locations to subsequent statements, starting with the value assigned to the program counter. To do addressing in relocatable sections, you use the 'program counter relative with displacement' addressing mode. The label value assignment is the same as for SECTION.

OFFSET Define offsets

Format: OFFSET <absexp>

To define a table of offsets via the DS directive beginning at the address <absexp>, you use the OFFSET directive. Symbols defined in an OFFSET table are kept internally, but no code-producing intructions or directives may appear. To terminate an OFFSET section, you use a RORG, OFFSET, SECTION, or END directive.

END End of program

Format: [<label>] END

The END directive tells the assembler that the source is finished, and the assembler ignores subsequent source statements. When the assembler encounters the END directive during the first pass, it begins the second pass. If, however, it detects an end-of-file before an END directive, it gives a warning message. If the label field is present, then the assembler assigns the value of the current program counter to the label before it executes the END directive.

Symbol Definition Directives

EQU Equate symbol value

Format: <label> EQU <exp>

The EQU directive assigns the value of the expression in the operand field to the symbol in the label field. The value assigned is permanent, so you may not define the label anywhere else in the program.

Note: Do not insert forward references within the expression.

EQUR Equate register value

Format: <label> EQUR <register>

This directive lets you equate one of the processor registers with a user symbol. Only the Address and Data registers are valid, so special symbols like SR, CCR, and USP are illegal here. The register is permanent, so you cannot define the label anywhere else in the program. The register must not be a forward reference to another EQUR statement. The assembler matches labels defined in this way without distinguishing upper and lower case.

REG Define register list

Format: <label> REG <register list>

The REG directive assigns a value to label that the assembler can translate into the register list mask format used in the MOVEM instruction. <register list> is of the form

R1 [-R2] [/R3 [-R4]]...

SET Set symbol value

Format: <label> SET <exp>

The SET directive assigns the value of the expression in the operand field to the symbol in the label field. SET is identical to EQU, apart from the fact that the assignment is temporary. You can always change SET later on in the program.

Note: You should not insert forward references within the expression or refer forward to symbols that you defined with SET.

Data Definition Directives

DC Define Constant

Format: [<label>] DC[.<size>] <list>

The DC directive defines a constant value in memory. It may have any number of operands, separated by commas (,). The values in the list must be capable of being held in the data location whose size is given by the size specifier on the directive. If you do not give a size specifier, DC assumes it is .W. If the size is .B, then there is one other data type that can be used: that of the ASCII string. This is an arbitrarily long series of ASCII characters, contained within quotation marks. As with ASCII literals, if you require a quotation mark in the string, then you must enter two. If the size is .W or .L, then the assembler aligns the data onto a word boundary.

DCB Define Constant Block

Format: [<label>] DCB[.<size>] <absexp>,<exp>

You use the DCB directive to set a number (given by <absexp>) of bytes, words, or longwords to the value of the expression <exp>. DCB.<size> n,exp is equivalent to repeating n times the statement DC.<size> exp.

DS Define Storage

Format: [<label>] DS[.<size>] <absexp>

To reserve memory locations, you use the DS directive. DS, however, does no initialization. The amount of space the assembler allocates depends on the data size (that you give with the size specifier on the directive), and the value of the expression in the operand field. The assembler interprets this as the number of data items of that size to allocate. As with DC, if the size specifier is .W or .L, DS aligns the space onto a word boundary. So, DS.W 0 has the effect of aligning to a word boundary only. If you do not give a size specifier, DS assumes a default of .W. See CNOP for a more general way of handling alignment.

Listing Control Directives

PAGE Page Throw

Format: PAGE

Unless paging has been inhibited, PAGE advances the assembly listing to the top of the next page. The PAGE directive does not appear on the output listing.

LIST Turn on Listing

Format: LIST

The LIST directive tells the assembler to produce the assembly listing file. Listing continues until it encounters either an END or a NOLIST directive. This directive is only active when the assembler is producing a listing file. The LIST directive does not appear on the output listing.

NOLIST Turn off Listing

Format: NOLIST
NOL

The NOLIST or NOL directive turns off the production of the assembly listing file. Listing ceases until the assembler encounters either an END or a LIST directive. The NOLIST directive does not appear on the program listing.

SPC Space Blank Lines

Format: SPC <number>

The SPC directive outputs the number of blank lines given by the operand field, to the assembly listing. The SPC directive does not appear on the program listing.

NOPAGE Turn off Paging

Format NOPAGE

The NOPAGE directive turns off the printing of page throws and title headers on the assembly listing.

LLEN Set Line Length

Format: LLEN <number>

The LLEN directive sets the line length of the assembly listing file to the value you specified in the operand field. The value must lie between 60 and 132, and can only be set once in the program. The LLEN directive does not appear on the assembly listing. The default is 132 characters.

PLEN Set Page Length

Format: PLEN <number>

The PLEN directive sets the page length of the assembly listing file to the value you specified in the operand field. The value must lie between 24 and 100, and you can only set it once in the program. The default is 60 lines.

TTL Set Program Title

Format: TTL <title string>

The TTL directive sets the title of the program to the string you gave in the operand field. This string appears as the page heading in the assembly listing. The string starts at the first non-blank character after the TTL, and continues until the end of line. It must not be longer than 40 characters in length. The TTL directive does not appear on the program listing.

NOOBJ Disable Object Code Generation

Format: NOOBJ

The NOOBJ directive disables the production of the object code file at the end of assembly. This directive disables the production of the code file, even if you specified a file name when you called the assembler.

FAIL Generate a user error

Format: FAIL

The FAIL directive tells the assembler to flag an error for this input line.

FORMAT No action

Format: FORMAT

The assembler accepts this directive but takes no action on receiving it. FORMAT is included for compatibility with other assemblers.

NOFORMAT No action

Format: NOFORMAT

The assembler accepts this directive but takes no action on receiving it. NOFORMAT is included for compatibility with other assemblers.

Conditional Assembly Directives

CNOP Conditional NOP

Format: [<label>] CNOP <number>,<number>

This directive is an extension from the Motorola standard and allows a section of code to be aligned on any boundary. In particular, it allows any data structure or entry point to be aligned to a long word boundary.

The first expression represents an offset, while the second expression represents the alignment required for the base. The code is aligned to the specified offset from the nearest required alignment boundary. Thus

```
        CNOP        0,4
```

aligns code to the next long word boundary while

```
        CNOP        2,4
```

aligns code to the word boundary 2 bytes beyond the nearest long word aligned boundary.

IFEQ Assemble if expresion = 0
IFNE Assemble if expression <> 0
IFGT Assemble if expression > 0
IFGE Assemble if expression >= 0
IFLT Assemble if expression < 0
IFLE Assemble if expression <= 0

Format: IFxx <absexp>

You use the IFxx range of directives to enable or disable assembly, depending on the value of the expression in the operand field. If the condition is not TRUE (for example, IFEQ 2+1), assembly ceases (that is, it is disabled). The conditional assembly switch remains active until the assembler finds a matching ENDC statement. You can nest conditional assembly switches arbitrarily, terminating each level of nesting with a matching ENDC.

IFC Assemble if strings are identical
IFNC Assemble if strings are not identical

```
Format:   IFC      <string>,<string>
          IFNC     <string>,<string>
```

The strings must be a series of ASCII characters enclosed in single quotes, for example, 'FOO' or '' (the empty string). If the condition is not TRUE, assembly ceases (that is, it is disabled). Again the conditional assembly switch remains active until the assembler finds a matching ENDC statement.

IFD Assemble if symbol defined
IFND Assemble if symbol not defined

```
Format:   IFD      <symbol name>
          IFND     <symbol name>
```

Depending on whether or not you have already defined the symbol, the assembler enables or disables assembly until it finds a matching ENDC.

ENDC End conditional assembly

Format: ENDC

To terminate a conditional assembly, you use the ENDC directive, set up with any of the 8 IFxx directives above. ENDC matches the most recently encountered condition directive.

Macro Directives

MACRO Start a macro definition

Format: <label> MACRO

MACRO introduces a macro definition. ENDM terminates a macro definition. You must provide a label, which the assembler uses as the name of the macro; subsequent uses of that label as an operand expand the contents of the macro and insert them into the source code. A macro can contain any opcode, most assembler directives, or any previously defined macro. A plus sign (+) in the listing, marks any code generated by macro expansion. When you use a macro name, you may append a number of arguments, separated by commas. If the argument contains a space (for example, a string containing a space) then you must enclose the entire argument within < (less than) and > (greater than) symbols.

The assembler stores up and saves the source code that you enter (after a MACRO directive and before an ENDM directive) as the contents of the macro. The code can contain any normal source code. In addition, the symbol \ (backslash) has a special meaning. Backslash followed by a number n indicates that the value of the nth argument is to be inserted into the code. If the nth argument is omitted then nothing is inserted. Backslash followed by the symbol '@' tells the assembler to generate the text '.nnn', where nnn is the number of times the \@ combination it has encountered. This is normally used to generate unique labels within a macro.

You may not nest macro definitions, that is, you cannot define a macro within a macro, although you can call a macro you previously defined. There is a limit to the level of nesting of macro calls. This limit is currently set at ten.

Macro expansion stops when the assembler encounters the end of the stored macro text, or when it finds a MEXIT directive.

NARG Special symbol

Format: NARG

The symbol NARG is a special reserved symbol and the assembler assigns it the index of the last argument passed to the macro in the parameter list (even nulls). Outside of a macro expansion, NARG has the value 0.

ENDM Terminate a macro definition

Format: ENDM

This terminates a macro definition introduced by a MACRO directive.

MEXIT Exit from macro expansion

Format: MEXIT

You use this directive to exit from macro expansion mode, usually in conjunction with the IFEQ and IFNE directives. It allows conditional expansion of macros. Once it has executed the directive, the assembler stops expanding the current macro as though there were no more stored text to include.

External Symbols

XDEF Define an internal label as an external entry point

Format: XDEF <label> [,<label>...]

One or more absolute or relocatable labels may follow the XDEF directive. Each label defined here generates an external symbol definition. You can make references to the symbol in other modules (possibly from a high-level language) and satisfy the references with a linker. If you use this directive or XREF, then you cannot directly execute the code produced by the assembler.

XREF Define an external name

Format: XREF <label> [,<label>...]

One or more labels that must not have been defined elsewhere in the program follow the XREF directive. Subsequent uses of the label tell the assembler to generate an external reference for that label. You use the label as if it referred to an absolute or relocatable value depending on whether the matching XDEF referred to an absolute or relocatable symbol.

The actual value used is filled in from another module by the linker. The linker also generates any relocation information that may be required in order for the resulting code to be relocatable.

External symbols are normally used as follows. To specify a routine in one program segment as an external definition, you place a label at the start of the routine and quote the label after an XDEF directive. Another program may call that routine if it declares a label via the XREF directive and then jumps to the label so declared.

General Directives

INCLUDE Insert an external file

Format: INCLUDE "<file name>"

The INCLUDE directive allows the inclusion of external files into the program source. You set up the file that INCLUDE inserts with the string descriptor in the operand field. You can nest INCLUDE directives up to a depth of three, enclosing the file names in quotes as shown. INCLUDE is especially useful when you require a standard set of macro definitions or EQUs in several programs.

You can place the definitions in a single file and then refer to them from other programs with a suitable INCLUDE. It is often convenient to place NOLIST and LIST directives at the head and tail of files you intend to include via INCLUDE. AmigaDOS searches for the file specification first in the current directory, then in each subsequent directory in the list you gave in the -i option.

MASK2 No action

Format: MASK2

The assembler accepts the MASK2 directive, but it takes no action on receiving it.

IDNT Name program unit

Format: IDNT <string>

A program unit, which consists of one or more sections, must have a name. Using the IDNT directive, you can define a name consisting of a string optionally enclosed in double quotes. If the assembler does not find a IDNT directive, it outputs a program unit name that is a null string.

Chapter 4: The Linker

This chapter describes the AmigaDOS Linker. The AmigaDOS Linker produces a single binary load file from one or more input files. It can also produce overlaid programs.

Table of Contents

4.1 Introduction

4.2 Using the Linker

4.2.1 Command Line Syntax

4.2.2 WITH Files

4.2.3 Errors and Other Exceptions

4.2.4 MAP and XREF Output

4.3 Overlaying

4.3.1 OVERLAY Directive

4.3.2 References To Symbols

4.3.3 Cautionary Points

4.4 Error Codes and Messages

4.1 Introduction

ALINK produces a single binary output file from one or more input files. These input files, known as **object files**, may contain external symbol information. To produce object files, you use your assembler or language translator. Before producing the output, or **load file**, the linker resolves all references to symbols.

The linker can also produce a link map and symbol cross reference table.

Associated with the linker is an **overlay supervisor**. You can use the overlay supervisor to overlay programs written in a variety of languages. The linker produces load files suitable for overlaying in this way.

You can drive the linker in two ways:

1. as a **Command line**. You can specify most of the information necessary for running the linker in the command parameters.

2. as a **Parameter file**. As an alternative, if a program is being linked repetitively, you can use a parameter file to specify all the data for the linker.

These two methods can take three types of input files:

1. **Primary binary input**. This refers to one or more object files that form the initial binary input to the linker. These files are always output to the load file, and the primary input must not be empty.

2. **Overlay files**. If overlaying, the primary input forms the root of the overlay tree, and the overlay files form the rest of the structure.

3. **Libraries**. This refers to specified code that the linker incorporates automatically. Libraries may be resident or scanned. A **resident library** is a load file which may be resident in memory, or loaded as part of the 'library open' call in the operating system. A **scanned library** is an object file within an archive format file. The linker only loads the file if there are any outstanding external references to the library.

The linker works in two passes.

1. In the first pass, the linker reads all the primary, library and overlay files, and records the code segments and external symbol information. At the end of the first pass, the linker outputs the map and cross reference table, if required.

2. If you specify an output file, then the linker makes second pass through the input. First it copies the primary input files to the output, resolving symbol references in the process, and then it copies out the required library code segments in the same way. Note that the library code segments form part of the root of the overlay tree. Next, the linker produces data for the overlay supervisor, and finally outputs the overlay files.

In the first pass, after reading the primary and overlay input files, the linker inspects its table of symbols, and if there are any remaining unresolved references, it reads the files, if any, that you specified as the library input. The linker then marks any code segments containing external definitions for these unresolved references for subsequent inclusion in the load file. The linker only includes those library code segments that you have referenced.

4.2 Using the Linker

To use the linker, you must know the command syntax, the type of input and output that the linker uses, and the possible errors that may occur. This section attempts to explain these things.

4.2.1 Command Line Syntax.

The ALINK command has the following parameters:

```
ALINK [FROM | ROOT] files [TO file] [WITH file]
  [VER file] [LIBRARY | LIB files] [MAP file]
  [XREF file] [WIDTH n]
```

The keyword template is

```
"FROM=ROOT,TO/K,WITH/K,VER/K,LIBRARY=LIB/K,
MAP/K,XREF/K,WIDTH/K"
```

In the above, `file` means a single file name, 'files' means zero or more file names, separated by a comma or plus sign, and 'n' is an integer.

The following are examples of valid uses of the ALINK command:

```
ALINK a
ALINK ROOT a+b+c+d MAP map-file WIDTH 120
ALINK a,b,c TO output LIBRARY :flib/lib,obj/newlib
```

When you give a list of files, the linker reads them in the order you specify.

The parameters have the following meanings:

FROM: specifies the object files that you want as the primary binary input. The linker always copies the contents of these files to the load file to form part of the overlay root. At least one primary binary input file must be specified. ROOT is a synonym for FROM.

TO: specifies the destination for the load file. If this parameter is not given, the linker omits the second pass.

WITH: specifies files containing the linker parameters, for example, normal command lines. Usually you only use one file here, but, for completeness, you can give a list of files. Note that parameters on the command line override those in WITH files. You can find a full description of the syntax of these files in section 4.2.2 of this manual.

VER: specifies the destination of messages from the linker. If you do not specify VER, the linker sends all messages to the standard output (usually the terminal).

LIBRARY: specifies the files that you want to be scanned as the library. The linker includes only referenced code segments. LIB is a valid alternative for LIBRARY.

MAP: specifies the destination of the link map.

XREF: specifies the destination of the cross reference output.

WIDTH: specifies the output width that the linker can use when producing the link map and cross reference table. For example, if you send output to a printer, you may need this parameter.

4.2.2 WITH Files

WITH files contain parameters for the linker. You use them to save typing a long and complex ALINK command line many times.

A WITH file consists of a series of parameters, one per line, each consisting of a keyword followed by data. You can terminate lines with a semicolon (;), where the linker ignores the rest of the line. You can then use the rest of the line after the semicolon to include a comment. The linker ignores blank lines.

The keywords available are as follows:

FROM (or ROOT) files
TO file
LIBRARY files
MAP [file]
XREF [file]
OVERLAY
tree specification
#
WIDTH n

where 'file' is a single filename, 'files' is one or more filenames, '[file]' is an optional filename, and 'n' is an integer. You may use an asterisk symbol (*) to split long lines; placing one at the end of a line tells the printer to read the next line as a continuation line. If the filename after MAP or XREF is omitted, the output goes to the VER file (the terminal by default).

Parameters on the command line override those in a WITH file, so that you can make small variations on standard links by combining command line parameters and WITH files. Similarly, if you specify a parameter more than once in WITH files, the linker uses the first occurrence.

Note: In the second example below, this is true even if the first value given to a parameter is null.

Examples of WITH files and the corresponding ALINK calls:

```
ALINK WITH link-file
```

where 'link-file' contains

```
FROM    obj/main,obj/s
TO      bin/test
LIBRARY obj/lib
MAP
XREF    xo
```

is the same as specifying

```
ALINK FROM obj/main,obj/s TO bin/test
     LIBRARY obj/lib XREF xo
```

The command

```
ALINK WITH lkin LIBRARY ""
```

where 'lkin' contains

```
FROM    bin/prog,bin/subs
LIBRARY nag/fortlib
TO      linklib/prog
```

is the same as the command line

```
ALINK FROM bin/prog,bin/subs TO linklib.prog
```

Note: In the example above, the null parameter for LIBRARY on the command line overrides the value 'nag/fortlib' in the WITH file, and so the linker does not read any libraries.

4.2.3 Errors and Other Exceptions

Various errors can occur while the linker is running. Most of the messages are self-explanatory and refer to the failure to open files, or to errors in command or binary file format. After an error, the linker terminates at once.

There are a few messages that are warnings only. The most-important ones refer to undefined or multiply-defined symbols. The linker should not terminate after receiving a warning.

If any undefined symbols remain at the end of the first pass, the linker produces a warning, and outputs a table of such symbols. During the second pass, references to these symbols become references to **location zero.**

If the linker finds more than one definition of a symbol during the first pass, it puts out a warning, and ignores the later definition. The linker does not produce this message if the second definition occurs in a library file, so that you can replace library routines without it producing spurious messages. A serious error follows if the linker finds inconsistent symbol references, and linking then terminates at once.

Since the linker only uses the first definition of any symbol, it is important that you understand the following order in which files are read.

1. Primary (FROM or ROOT) input.
2. Overlay files.
3. LIBRARY files.

Within each group, the linker reads the files in the order that you specify in the file list. Thus definitions in the primary input override those in the overlay files, and those in the libraries have lowest priority.

4.2.4 MAP and XREF Output

The link map, which the linker produces after the first pass, lists all the code segments that the linker output to the load file in the second pass, in the order that they must be written.

For each code segment, the linker outputs a header, starting with the name of the file (truncated to eight letters), the code segment reference number, the type (that is, data, code, bss, or COMMON), and size. If the code segment was in an overlay file, the linker also gives the overlay level and overlay ordinate.

After the header, the linker prints each symbol defined in the code segment, together with its value. It prints the symbols in ascending order of their values, appending an asterisk (*) to absolute values.

The value of the WIDTH parameter determines the number of symbols printed per line. If this is too small, then the linker prints one symbol on each line.

The cross reference output also lists each code segment, with the same header as in the map.

The header is followed by a list of the symbols with their references. Each reference consists of a pair of integers, giving the offset of the reference and the number of the code segment in which it occurs. The code segment number refers to the number given in each header.

4.3 Overlaying

The automatic overlay system provided by the linker and the overlay supervisor allows programs to occupy less memory when running, without any alterations to the program structure.

When using overlaying, you should consider the program as a **tree** structure. That is, with the **root** of the tree as the primary binary input, together with library code segments and COMMON blocks. This root is always resident in memory. The overlay files then form the other nodes of the tree, according to specifications in the OVERLAY directive.

The output from the linker when overlaying, as in the usual case, is a single binary file, which consists of all the code segments, together with information giving the location within the file of each node of the overlay tree. When you load the program only the root is brought into memory. An overlay supervisor takes care of loading and unloading the overlay segments automatically. The linker includes this overlay supervisor in the output file produced from an link using overlays. The overlay supervisor is invisible to the program running.

4.3.1 OVERLAY Directive

To specify the tree structure of a program to the linker, you use the OVERLAY directive. This directive is exceptional in that you can only use it in WITH files. As with other parameters, the linker uses the first OVERLAY directive you give it.

The format of the directive is

```
OVERLAY
Xfiles
•
•
•
#
```

Note: The overlay directive can span many lines. The linker recognizes a hash sign (#) or the end-of-file as a terminator for the directive.

Each line after OVERLAY specifies one node of the tree, and consists of a count X and a file list.

The level of a node specifies its 'depth' in the tree, starting at zero, which is the level of the root. The count, X, given in the directive, consists of zero or more asterisks, and the overlay level of the node is given by X+1.

As well as the level, each node other than the root has an **ordinate** value. This refers to the order in which the linker should read the descendents of each node, and starts at 1, for the first 'offspring' of a parent node.

Note: There may be nodes with the same level and ordinate, but with different parents.

While reading the OVERLAY directive, the linker remembers the current level, and, for each new node, compares the level specified with this value. If less, then the new node is a descendent of a previous one. If equal, the new node has the same parent as the current one. If greater, the new node is a direct descendant of the current one, and so the new level must be one greater than the current value.

A number of examples may help to clarify this:

```
Directive   Level   Ordinate     Tree

 OVERLAY                         ROOT
 a            1        1          /|\
 b            1        2         a b c
 c            1        3
 #

 OVERLAY                         ROOT
 a            1        1          /\
 b            1        2         a  b
 *c           2        1            /|
 *d           2        2           c d
 #
```

Figure 4-A

```
OVERLAY                         -ROOT-
a            1        1         / /|\  \
b            1        2        / / | \  \
*c           2        1       a b  e  f  l
*d           2        2        /|    /|\
e            1        3       c d   g h k
f            1        4              /|
*g           2        1             i j
*h           2        2
**i          3        1
**j          3        2
*k           2        3
l            1        5
#
```

(continuation of Figure 4-A)

The level and ordinate values given above refer to the node specified on the same line. Note that all the files given in the examples above could have been file lists. Single letters are for clarity. For example, Figure 4-B

```
ROOT      bin/mainaaa
OVERLAY
bin/mainbbb,bin/mainccc,bin/mainddd
*bin/makereal
*bin/trbblock,bin/transint,bin/transr*
 bin/transri
bin/outcode
#
```

Figure 4-B

specifies the tree in the following figure:

```
              bin/mainaaa
                /\
               /  \
              /    \
             /      \
            /        \
           /          \
       bin/mainbbb  bin/outcode
       bin/mainccc
       bin/mainddd
         /\
        /  \
       /    \
      /      \
     /        \
    /          \
bin/makereal bin/trbblock
             bin/transint
             bin/transr
             bin/transri
```

Figure 4-C

During linking, the linker reads the overlay files in the order you specified in the directive, line by line. The linker preserves this order in the map and cross reference output, and so you can deduce the exact tree structure from the overlay level and ordinate the linker prints with each code segment.

4.3.2 References To Symbols

While linking an overlaid program, the linker checks each symbol reference for validity.

Suppose that the reference is in a tree node R, and the symbol in a node S. Then the reference is legal if one of the following is true.

(a) R and S are the same node.
(b) R is a descendent of S.
(c) R is the parent of S.

References of the third type above are known as **overlay references**. In this case, the linker enters the overlay supervisor when the program is run. The overlay supervisor then checks to see if the code segment containing the symbol is already in memory. If not, first the code segment, if any, at this level, and all its descendents are unloaded, and then the node containing the symbol is brought into memory. An overlaid code segment returns directly to its caller, and so is not unloaded from memory until another node is loaded on top of it.

For example, suppose that the tree is:

```
    A
   /|
  / |
  B C
 /|\
/ | \
D E F
```

When the linker first loads the program, only A is in memory. When the linker finds a reference in A to a symbol in B, it loads and enters B. If B in turn calls D then again a new node is loaded. When B returns to A, both B and D are left in memory, and the linker does not reload them if the program requires them later. Now suppose that A calls C. First the linker unloads the code segments that it does not require, and which it may overwrite. In this case, these are B and D. Once it has reclaimed the memory for these, the linker can load C.

Thus, when the linker executes a given node, all the node's 'ancestors', up to the root are in memory, and possibly some of its descendents.

4.3.3 Cautionary Points

The linker assumes that all overlay references are jumps or subroutine calls, and routes them through the overlay supervisor. Thus, you should not use overlay symbols as data labels.

Try to avoid impure code when overlaying because the linker does not always load a node that is fresh from the load file.

The linker gives each symbol that has an overlay reference an **overlay number**. It uses this value, which is zero or more, to construct the overlay supervisor entry label associated with that symbol. This label is of the form 'OVLYnnnn', where nnnn is the overlay number. You should not use symbols with this format elsewhere.

The linker gathers together all program sections with the same section name. It does this so that it can then load them continuously in memory.

4.4 Error Codes and Messages

These errors should be rare. If they do occur, the error is probably in the compiler and not in your program. However, you should first check to see that you sent the linker a proper program (for example, an input program must have an introductory program unit that tells the linker to expect a program).

Invalid Object Modules

2	Invalid use of overlay symbol.
3	Invalid use of symbol
4	Invalid use of common
5	Invalid use of overlay reference
6	Non-zero overlay reference
7	Invalid external block relocation
8	Invalid bss relocation
9	Invalid program unit relocation
10	Bad offset during 32 bit relocation
11	Bad offset during 6/8 bit relocation
12	Bad offset with 32 bit reference
13	Bad offset with 6/8 bit reference
14	Unexpected end of file
15	Hunk.end missing
16	Invalid termination of file
17	Premature termination of file
18	Premature termination of file

Internal Errors

19	Invalid type in hunk list
20	Internal error during library scan
21	Invalid argument freevector
22	Symbol not defined in second pass

Appendix A: Console Input and Output on the Amiga

Note: Throughout this appendix, the characters "<CSI>" represent the "Control Sequence Introducer. For output, you may either use the two character sequence Esc-[or the one byte value $9B (hex). For input, you receive $9B's.

Introduction

This appendix describes several ways to do console (keyboard and screen) input and output on the Amiga. You can open the console as you would any other AmigaDOS file (with "*", "CON:", "RAW:") or do direct calls to console.library. The advantages of using each are listed below:

* "Star" does not open any windows; it just uses the existing CLI window. You do not receive any complex character sequences. You do receive lowercase letters a-z, uppercase letters A-Z, numbers, ASCII special symbols, and control characters. Basically, if a teletype can generate the character with a single keystroke, you can receive it. In addition to these characters, you can receive each of them with the high-order bit set ($80-$FF). Line editing is also performed for you. This means AmighaDOS accepts <BackSpace> and CRTL-X for character and line deletions. You do not have to deal with these. Any <CSI> sequence is swallowed for you as well as control characters: C, D, E, F, H, and X. Any <CR> or CTRL-M characters are converted to CTRL-J (new-line).

CON: Is just like "*" except that you also get to define a new window.

RAW: The simple case: With RAW: (as compared to CON:) you lose the line editing functions and you gain access to the function and arrow keys. These are sent as sequences of characters which you must parse in an intelligent manner.

The "complex" cases: By issuing additional commands to the console processor (by doing writes to RAW:), you can get even more detailed information. For example, you can request key press and release information or data on mouse events. See "Selection of RAW Input Events" below for details on requesting this information.

console.library:
With thıs method, you have full control over the console device. You may change the KeyMap to one of your own design and completely "redesign" your keyboard.

Helpful AmigaDOS Commands

Two very helpful AmigaDOS commands let you play with these functions. The first:

```
TYPE RAW:10/10/100/30/ opt h
```

accepts input from a RAW: window and displays the results in hex and ASCII. If you want to know for sure what characters the keyboard is sending, this command provides a very simple way.

The second:

```
COPY "RAW:10/10/100/30/RAW Input" "RAW:100/10/200/100/RAW Output"
```

lets you type sequences into the input window and watch the cursor movement in the output window. COPY cannot detect end of file on RAW: input, so you have to reboot when you are finished with this command.

CON Keyboard Input

If you read from the CON: device, the keyboard inputs are preprocessed for you.

You get the ASCII characters like "B". Most normal text gathering programs read from the CON: device. Special programs like word processors and music keyboard programs use RAW:.

To generate the international and special characters at the keyboard, you can press either ALT key. This sets the high bit of the ASCII code returned for the key pressed.

Generating $FF (umlaut y) is a special case. If it followed the standard convention, it would be generated by ALT-DEL. But since the ASCII code <Del> (hex 7F) is not generally a printable character and it is our philosophy that Alt-non-printing character should not generate a printing character, we have substituted ALT-numeric pad "-".

Table A-1 lists the characters you can display on the Amiga. The characters NBSP (non-break space) and SHY (soft hyphen) are used to render a space and hyphen in text processing with additional meaning about the properties of the character.

Table A-1: International Character Code

				b.	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
				b.	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
				b.	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
				b.	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
b.	b.	b.	b.		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
0	0	0	0	00	▨	▨	SP	0	@	P	`	p	▨	▨	NBSP	°	À	Đ	à	ð
0	0	0	1	01	▨	▨	!	1	A	Q	a	q	▨	▨	¡	±	Á	Ñ	á	ñ
0	0	1	0	02	▨	▨	"	2	B	R	b	r	▨	▨	¢	²	Â	Ò	â	ò
0	0	1	1	03	▨	▨	#	3	C	S	c	s	▨	▨	£	³	Ã	Ó	ã	ó
0	1	0	0	04	▨	▨	$	4	D	T	d	t	▨	▨	¤	´	Ä	Ô	ä	ô
0	1	0	1	05	▨	▨	%	5	E	U	e	u	▨	▨	¥	µ	Å	Õ	å	õ
0	1	1	0	06	▨	▨	&	6	F	V	f	v	▨	▨	¦	¶	Æ	Ö	æ	ö
0	1	1	1	07	▨	▨	'	7	G	W	g	w	▨	▨	§	·	Ç	▩	ç	▩
1	0	0	0	08	▨	▨	(	8	H	X	h	x	▨	▨	¨	¸	È	Ø	è	ø
1	0	0	1	09	▨	▨	)	9	I	Y	i	y	▨	▨	©	¹	É	Ù	é	ù
1	0	1	0	10	▨	▨	*	:	J	Z	j	z	▨	▨	ª	º	Ê	Ú	ê	ú
1	0	1	1	11	▨	▨	+	;	K	[	k	{	▨	▨	«	»	Ë	Û	ë	û
1	1	0	0	12	▨	▨	,	<	L	\	l	|	▨	▨	¬	¼	Ì	Ü	ì	ü
1	1	0	1	13	▨	▨	-	=	M	]	m	}	▨	▨	SHY	½	Í	Ý	í	ý
1	1	1	0	14	▨	▨	.	>	N	^	n	~	▨	▨	®	¾	Î	Þ	î	þ
1	1	1	1	15	▨	▨	/	?	O	_	o	▨	▨	▨	¯	¿	Ï	ß	ï	ÿ

Note: AmigaDOS uses CON: input for the CLI and most other commands. When it does this, it filters out ALL of the function key and cursor key inputs. Programs that run under AmigaDOS can (and some do) still open the RAW: console handler and process function key input.

CON Screen Output

CON: screen output is just like RAW: screen output except that <LF> (hex 0A) is translated into a new-line character. The net effect is that the cursor moves to the first column of the next line whenever a <LF> is displayed.

RAW Screen Output

ANSI x3.64 CODES SUPPORTED For writing text to the display:

Independent Control Functions (no introducer):

Ctrl	Hex	Name	Definition	
H	08	BS	BACKSPACE	Move the cursor left 1 column
I	09	TAB	TAB	Move right 1 column
J	0A	LF	LINE FEED	
K	0B	VT	VERTICAL TAB	Move cursor up 1, scroll if necessary
L	0C	FF	FORM FEED	Clear the screen
M	0D	CR	CARRIAGE RETURN	Move to first column
N	0E	SO	SHIFT OUT	Set MSB of each character before displaying
O	0F	SI	SHIFT IN	Undo SHIFT OUT
[	1B	ESC	ESCAPE	See below

Precede the following characters with <ESC> to perform the indicated actions.

Chr	Name	Definition
c	RIS	RESET TO INITIAL STATE

Precede the following characters with <Esc> or press CTRL-ALT and the letter to perform the indicated actions.

Hex	Chr	Name	Definition
845tD	IND	INDEX:	move the active position down one line
85	E	NEL	NEXT LINE:
8D	M	RI	REVERSE INDEX:
9B	[	CSI	CONTROL SEQUENCE INTRODUCER: see next list

Control Sequences (introduced by <CSI>) with parameters. The first character in the following table (under the <CSI> column) represents the number of allowable parameters, as follows:

"0" indicates no parameters allowed.
"1" indicates 0 or 1 numeric parameters.
"2" indicates 2 numeric parameters. ('14;94')
"3" indicates any number of numeric parameters, separated by semicolons.
"4" indicates exactly 4 numeric parameters.
"8" indicates exactly 8 numeric parameters.

<CSI>	Name	Definition	
1 @	ICH	INSERT CHARACTER	Inserts 1 or more spaces, shifting the remainder of the line to the right.
1 A	CUU	CURSOR UP	
1 B	CUD	CURSOR DOWN	
1 C	CUF	CURSOR FORWARD	
1 D	CUB	CURSOR BACKWARD	
1 E	CNL	CURSOR NEXT LINE	Down n lines to column 1
1 F	CPL	CURSOR PRECEDING LINE	Up n lines to column 1
2 H	CUP	CURSOR POSITION	"<CSI>row;columnH"
1 J	ED	ERASE IN DISPLAY	(only to end of display)
1 K	EL	ERASE IN LINE	(only to eol)
1 L	IL	INSERT LINE	Inserts a blank line BEFORE the line containing the cursor.
1 M	DL	DELETE LINE	Removes the current line. Moves all lines below up by one. Blanks the bottom line.
1 P	DCH	DELETE CHARACTER	
2 R	CPR	CURSOR POSITION REPORT	(in Read stream only) Format of report: "<CSI>row;columnR"
1 S	SU	SCROLL UP	Removes line from top of screen. Moves all other lines up one. Blanks last line.
1 T	SD	SCROLL DOWN	Removes line from bottom of screen. Moves all other lines down one. Blanks top line.
3 h	SM	SET MODE	<CSI>20h causes RAW: to convert <LF> to <new-line> on output.
3 l	RM	RESET MODE	<CSI>20l undoes SET MODE 20
3 m	SGR	SELECT GRAPHIC RENDITION	
1 n	DSR	DEVICE STATUS REPORT	

The following are not ANSI standard sequences; rather, they are private Amiga sequences.

1	t	aSLPP SET PAGE LENGTH
1	u	aSLL SET LINE LENGTH
1	x	aSLO SET LEFT OFFSET
1	y	aSTO SET TOP OFFSET
3	{	aSRE SET RAW EVENTS
8	|	aIER INPUT EVENT REPORT (read)
3	}	aRRE RESET RAW EVENTS
1	~	aSKR SPECIAL KEY REPORT (read)
1	p	aSCR SET CURSOR RENDITION
		<Esc> p turns the cursor off
0	q	aWSR WINDOW STATUS REQUEST
4	r	aWBR WINDOW BOUNDS REPORT (read)

Examples:

Move cursor right by 1:

```
<CSI>C or <Tab> or <CSI>1C
```

Move cursor right by 20:

```
<CSI>20C
```

Move cursor to upper left corner (home):

```
<CSI>H  or <CSI>1;1H or <CSI>;1H or <CSI>1;H
```

Move cursor to the forth column of the first line of the window:

```
<CSI>1;4H or <CSI>;4H
```

Clear the screen:

```
<FF> or CTRL-L       {clear screen character} or
<CSI>H<CSI>J         {home and clear to end of screen} or
<CSI>H<CSI>23M       {home and delete 23 lines} or
<CSI>1;1H<CSI>23L    {home and insert 23 lines}
```

RAW Keyboard Input

Reading input from the RAW: console device returns an ANSI x3.64 standard byte stream. This stream may contain normal characters and/or RAW input event information. You may also request other RAW input events using the SET RAW EVENTS (aSRE) and RESET RAW EVENTS (aRRE) control sequences discussed below. See "Selection of RAW input events" for details.

If you issue a RAW input request and there is no pending input, the read command waits until some input is received. You can test for characters pending by doing "WaitforChar" requests.

In the default state, the function and arrow keys cause the following sequences to be sent to your process:

Key	Unshifted Sends	Shifted Sends
F1	<CSI>0~	<CSI>10~
F2	<CSI>1~	<CSI>11~
F3	<CSI>2~	<CSI>12~
F4	<CSI>3~	<CSI>13~
F5	<CSI>4~	<CSI>14~
F6	<CSI>5~	<CSI>15~
F7	<CSI>6~	<CSI>16~
F8	<CSI>7~	<CSI>17~
F9	<CSI>8~	<CSI>18~
F10	<CSI>9~	<CSI>19~
HELP	<CSI>?~	<CSI>?~ (same)

Arrow keys:

Up	<CSI>A	<CSI>T~
Down	<CSI>B	<CSI>S~
Left	<CSI>C	<CSI> A~ (note space)
Right	<CSI>D	<CSI> @~ (note space)

Selection of RAW Input Events:

If you are using RAW by default, you get the ANSI data and control sequences mentioned above. If this does not give you enough information about input events, you can request additional information from the console driver.

If, for example, you need to know when each key is pressed and released, you would request "RAW keyboard input." This is done by writing "<CSI>1{" to the console. The following is a list of valid RAW input requests:

RAW Input Event Types

Request Number	Description	
0	nop	Used internally
1	RAW keyboard input	
2	RAW mouse input	
3	Event	Sent whenever your window is made active
4	Pointer position	
5	(unused)	
6	Timer	
7	Gadget pressed	
8	Gadget released	
9	Requester activity	
10	Menu numbers	
11	Close Gadget	
12	Window resized	
13	Window refreshed	
14	Preferences changed (not yet implemented)	
15	Disk removed	
16	Disk inserted	

If you select any of these events, you start to get information about the events in the following form:

```
<CSI><class>;<subclass>;<keycode>;<qualifiers>;<x>;<y>;
<seconds>;<microseconds>|
```

<CSI> is a one byte field. It is the Control Sequence Introducer, 9B hex.

<class> is the RAW input event type, from the above table.

<subclass> is not currently used and is always zero (0).

<keycode> indicates which key number was pressed (see Figure A-1 and Table A-2). This field can also be used for mouse information.

The <qualifiers> field indicates the state of the keyboard and system. The qualifiers are defined as follows:

Bit	Mask	Key	
0	0001	left shift	
1	0002	right shift	
2	0004	capslock	* special, see below
3	0008	control	
4	0010	left alt	
5	0020	right alt	
6	0040	left Amiga key pressed	
7	0080	right Amiga key pressed	
8	0100	numeric pad	
9	0200	repeat	
10	0400	interrupt	Not currently used
11	0800	multi broadcast	This (active) or all windows
12	1000	left mouse button	
13	2000	right mouse button	
14	4000	middle mouse button (not available on std mouse)	
15	8000	relative mouse	Indicates mouse coordinates are relative, not absolute

The CAPS LOCK key is handled in a special manner. It only generates a keycode when it is pressed, not when it is released. However, the up and down bit (80 hex) is still used and reported If pressing the CAPS LOCK key turns on the LED, then key code 62 (CAPS LOCK pressed) is sent. If pressing the caps lock key extinguishes the LED, then key code 190 (CAPS LOCK released) is sent. In effect, the keyboard reports this key being held down until it is struck again.

The <seconds> and <microseconds> fields are system time stamp taken at the time the event occurred. These values are stored as long-words by the system and as such could (theoretically) reach 4 billion.

With RAW: keyboard input, selected keys no longer return a simple 1 character "A" to "Z" but rather return raw keycode reports with the following form:

```
<CSI>1;0;<keycode>;<qualifiers>;0;0;<secs>;<microsecs>|
```

For example, if the user pressed and released the "B" key with the left SHIFT and right Amiga keys also pressed, you might receive the following data:

```
<CSI>1;0;35;129;0;0;23987;99|
<CSI>1;0;163;129;0;0;24003;18|
```

The "0;0;" fields are for not used for keyboard input but are, rather used if you select mouse input. For mouse input, these fields would indicate the X and Y coordinates of the mouse.

The <keycode> field is an ASCII decimal value representing the key pressed or released. Adding 128 to the pressed key code results in the released keycode. Figure A-1 lets you convert quickly from a key to its keycode. Table A-1 lets you convert quickly from a keycode to a key.

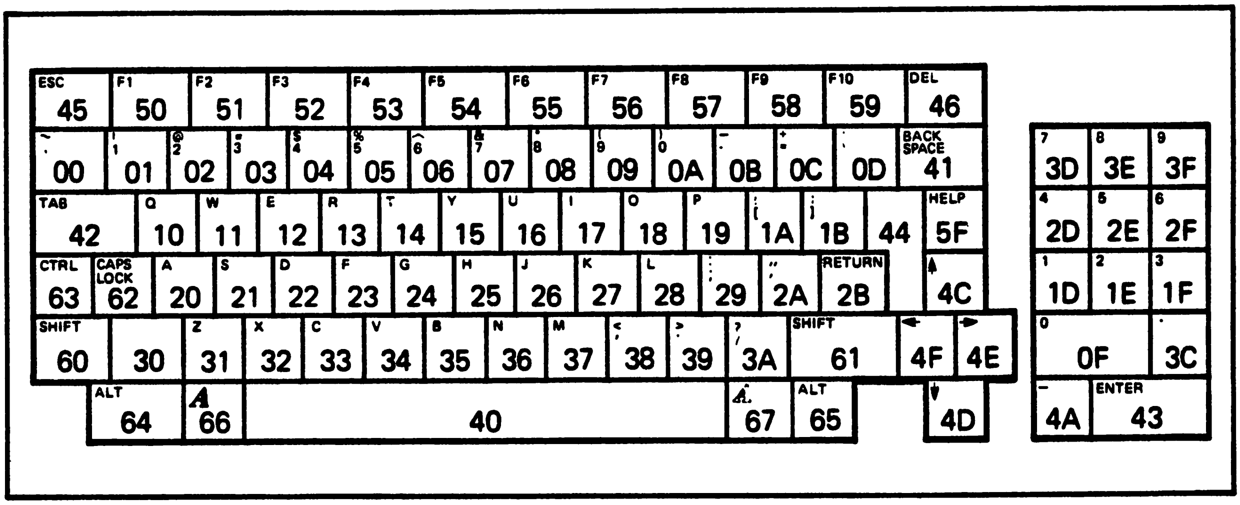


Figure A-1, reduced copy of keyboard template

Table A-2: Converting from Keycodes to Keys

The default values given in the following table correspond to:

1) The values the CON: device returns when these keys are pressed, and

2) The keycaps as shipped with the standard American keyboard.

Table A-2

Raw	Unshifted	Shifted
Key	Default	Default
Number	Value	Value
00	` (Accent grave)	~ (tilde)
01	1	!
02	2	@
03	3	#
04	4	$
05	5	%
06	6	^
07	7	&
08	8	*
09	9	(
0A	0	)
0B	- (Hyphen)	_ (Underscore)
0C	=	+
0D	\	|
0E	(undefined)	
0F	0	0 (Numeric pad)
10	Q	q
11	W	w
12	E	e
13	R	r
14	T	t
15	Y	y
16	U	u
17	I	i
18	O	o
19	P	p
1A	{	[
1B	}	]
1C	(undefined)	
1D	1	1 (Numeric pad)
1E	2	2 (Numeric pad)
1F	3	3 (Numeric pad)
20	A	a
21	S	s
22	D	d
23	F	f
24	G	g
25	H	h
26	J	j
27	K	k
28	L	l

Raw Key Number	Unshifted Default Value	Shifted Default Value
29	:	;
2A	"	' (single quote)
2B	(RESERVED)	(RESERVED)
2C	(undefined)	
2D	4	4 (Numeric pad)
2E	5	5 (Numeric pad)
2F	6	6 (Numeric pad)
30	(RESERVED)	(RESERVED)
31	Z	z
32	X	x
33	C	c
34	V	v
35	B	b
36	N	n
37	M	m
38	<	, (comma)
39	>	. (period)
3A	?	/
3B	(undefined)	
3C	.	. (Numeric pad)
3D	7	7 (Numeric pad)
3E	8	8 (Numeric pad)
3F	9	9 (Numeric pad)
40	Space	
41	BACKSPACE	
42	TAB	
43	ENTER	ENTER (Numeric pad)
44	RETURN	
45	Escape	(Esc)
46	DEL	
47	(undefined)	
48	(undefined)	
49	(undefined)	
4A	-	- (Numeric Pad)
4B	(undefined)	
4C	Cursor Up	Scroll down
4D	Cursor Down	Scroll up
4E	Cursor Forward	Scroll left
4F	Cursor Backward	Scroll right
50	F1	<CSI>10~
51	F2	<CSI>11~
52	F3	<CSI>12~
53	F4	<CSI>13~

Raw Key Number	Unshifted Default Value	Shifted Default Value
54	F5	<CSI>14~
55	F6	<CSI>15~
56	F7	<CSI>16~
57	F8	<CSI>17~
58	F9	<CSI>18~
59	F10	<CSI>19~
5A	(undefined)	
5B	(undefined)	
5C	(undefined)	
5D	(undefined)	
5E	(undefined)	
5F	Help	
60	SHIFT (left of space bar)	
61	SHIFT (right of space bar)	
62	Caps Lock	
63	Control	
64	Left Alt	
65	Right Alt	
66	"Amiga" (left of space bar)	
67	"Amiga" (right of space bar)	
68	Left Mouse Button (not converted)	Inputs are only for the
69	Right Mouse Button (not converted)	mouse connected to Intuition,
6A	Middle Mouse Button (not converted)	currently "gameport" one.
6B	(undefined)	
6C	(undefined)	
6D	(undefined)	
6E	(undefined)	
6F	(undefined)	
70-7F	(undefined)	
80-F8	Up transition (release or unpress key) of one of the above keys. 80 for 00, F8 for 7F.	
F9	Last keycode was bad (was sent in order to resync)	
FA	Keyboard buffer overflow.	
FB	(undefined, reserved for keyboard processor catastrophe)	
FC	Keyboard self-test failed.	

Raw Key Number	Unshifted Default Value
FD	Power-up key stream start. Keys pressed or stuck at power-up are sent between FD and FE.
FE	Power-up key stream end.
FF	(undefined, reserved)
FF	Mouse event, movement only, No button change. (not converted)

Notes about the preceding table:

1) "(undefined)" indicates that the current keyboard design should not generate this number. If you are using "SetKeyMap" to change the key map, the entries for these numbers must still be included.

2) The "(not converted)" refers to mouse button events. You must use the sequence "<CSI>2{" to inform the console driver that you wish to receive mouse events; otherwise, these are not transmitted.

3) "(RESERVED)" indicates that these keycodes have been reserved for non-US keyboards. The "2B" code key is between the double quote and return keys. The "30" code key is between the SHIFT and Z keys.

" (double quote) 3.2, 3.13
\ (backslash) 3.2, 3.20
_ (underscore) 3.6
~ Logical NOT 3.6
(hash, or sharp sign) 4.6
& AND 3.6
' 3.8, 3.19
* 2.8, 3.3, 3.6, 3.7, 4.3, 4.5
+ 3.2, 3.6, 3.20
, 3.2, 3.5, 3.15, 3.20
- Monadic minus 3.6
- Subtract 3.6
-c 3.2
-h 3.2
-i 3.2, 3.22
-l 3.2
-o 1.8, 3.2
-v 3.2
. (period) 3.5, 3.6
.L (longword) 3.8, 3.15, 3.16
.W (word) 3.8, 3.15, 3.16
/ 3.6
68000 assembler 1.1, 3.1
: (colon) 3.4
; (semicolon) 3.3
< 1.3, 3.20
<< Left shift 3.6
> 1.3, 3.20
>> Right shift 3.6
@ 3.20

A Absolute symbols 3.6
A0-A7 1.2, 1.3, 3.1, 3.8
Absolute expression 3.8
Absolute origin 3.11
Absolute symbols 3.6, 3.7, 3.22
Add (+) 3.6
Address mode 3.8, 3.9
Address registers 3.1, 3.8, 3.10, 3.14
Address variant 3.10
Address, byte 3.1
Addressing, indexed 3.1
Alignment 3.15, 3.16, 3.18, 3.19
ALINK 1.10, 4.1-10
Allocated memory, return 1.4
Amiga structure definitions 1.1
AmigaDOS library 1.2, 1.3
Argument list 1.3
Arguments 2.1, 3.20
ASCII characters 3.19
ASCII files 1.8, 1.9
ASCII hex file 1.9
ASCII literal 3.8
ASCII string 3.15
ASSEM 3.2, 3.1-23
Assemble if <condition> 3.12, 3.19
Assemble program 1.2
Assembler 1.1, 1.2, 1.3, 1.4, 1.5, 1.10, 3.1-23
Assembler command line 3.2
Assembler directives 3.3, 3.5, 3.11-23
Assembler, options 3.2, 3.3
Assembly control 3.11, 3.13
Assembly listing 3.17, 3.16
Assembly statistics 3.2
Assign permanent register value 3.11
Assign permanent value 3.11
Assign temporary value 3.11
Automatic overlaying 4.5

B Byte size (8-bits) 3.5
Backing up 1.1
BillBoard 1.7
Binary 3.8
Binary code file 1.3
Binary files, transfer 1.2
Binary load file 1.9
Binary to hex, converting 1.10
Binload 1.7, 1.8
Boolean returns 2.1
Branches 3.1
BSS 3.13
Byte identification in memory 3.1
Byte operations 3.1

C 1.1, 1.2, 1.3, 1.4, 1.5, 1.10
C compiler 1.5, 1.10
C option (assem) 3.3
C programming language - see C
Case (Upper/lower) 2.1
Case distinction (assem option) 3.3, 3.7, 3.14
CCR 3.7, 3.14
Character files, transfer 1.2
Characters, return 2.13
CLI 1.3, 1.4, 1.5, 1.10, 2.16
Close 2.2
Close file for I/O 1.4, 2.2
CNOP 3.12, 3.16, 3.18
CODE 3.13
Code segment 4.1, 4.2, 4.5, 4.8, 4.9
Code, load/unload 1.4
Comma - see ,
Command line input (ALINK) 4.1
Command Line Interface - see CLI
Command line, assembler 3.2
Comment field 3.5
Comment or blank lines 3.3
Comment, set file or directory 2.11
COMMON blocks 4.5
Compile program 1.2
Compiler 1.1, 1.5, 1.10

Compiling C - see C compiler
CON: 2.8
Conditional assembly 3.12, 3.18, 3.19, 3.20
Conditional directive 3.20
Conditional expansion of macros 3.21
Conditional NOP 3.12, 3.18
Constant value assignment 3.15
Constants, define 3.11
Control codes 1.8
Control of listing 3.12
Convert 1.9, 1.10
Converting binary to hex 1.9, 1.10
Create a new directory 2.2
Create a new process 2.14
CreateDir 2.2
CreateProc 2.14
Cross compilation 1.10
Cross development 1.1, 1.2, 1.5, 1.10
Cross reference output 4.3, 4.8
Cross reference table 3.3, 4.1, 4.3
CTRL-C 1.8
CTRL-D 1.8
CTRL-E 1.8
CTRL-F 1.8
Current directory 3.2
Current program counter 3.7
Current window 2.8
CurrentDir 2.3

D option 3.3
D0-D7 1.3, 1.4, 3.1, 3.8
DATA 3.13
Data definition 3.11, 3.15
Data labels 4.9
Data registers 3.1, 3.8, 3.14
Data size 3.5, 3.16
Data type 3.15
Date, internal format 2.15
DateStamp 2.15
DC 3.11, 3.15
DCB 3.11, 3.15
Decimal 3.7
Default line length 3.17
Default output, change 1.8
Default page length 3.17
Define constant block 3.11, 3.15
Define external name 3.12, 3.22
Define internal label as external entry point 3.21
Define macro name 3.12
Define offsets 3.11, 3.14
Define register list 3.15
Define storage 3.11, 3.16
Delay 2.15
DeleteFile 2.3
Device names 2.8
DeviceProc 2.7, 2.15
Diagnostic messages 3.2
Direct access of memory 3.1
Directive names 3.4, 3.7
Directives 3.11-23
Directory entry, examine next 2.5
Directory list (for file inclusion) 3.2, 3.22
Directory lock 2.2, 2.3, 2.8, 2.15
Directory, create a new 2.2
Directory, delete 2.3
Directory, examine 2.4, 2.5
Directory, make current 2.3
Directory, parent 2.9
Directory, rename 2.10
Directory, unlock 2.12
Disable assembly 3.19
Disable object code generation 3.18
Disable object code output 3.12
Disk information, return 2.6
Divide / 3.6
Documentation, required 1.1
Download 1.2, 1.8, 1.9
Downloading 1.1, 1.2, 1.8, 1.9, 1.10
DS 3.11, 3.14, 3.16
Dumping local labels 3.3
Duplicate lock 2.4
DupLock 2.4

Empty string 3.19
Enable assembly 3.19
Encoding programs (Assembler) 3.3
END 3.11, 3.14, 3.16
End conditional assembly - see ENDC
End macro definition - see ENDM
End of program - see END
ENDC 3.12, 3.19, 3.20
ENDCLI 2.17
ENDM 3.12, 3.20, 3.21
Entry, program 1.2
EQU 3.7, 3.11, 3.14, 3.15, 3.22
Equate register value - see EQUR
Equate symbol value - see EQU
EQUR 3.7, 3.11, 3.14
Error code 1.3
Error information 2.7
Error messages 3.2, 4.4
Error return 1.3
Error, user 3.18
Examine 2.4, 2.5
Examine directory or file 2.4, 2.5
Examine next directory entry 2.5
Example of OVERLAY 4.6
Example of valid uses of ALINK 4.2
Exceptions 4.4

Exclusive write lock 2.8
Executable instructions 3.3, 3.4
Executable program, creating an 1.2
EXECUTE 1.4,
Execute 2.16, 2.17
Exit 1.4, 2.16
Exit from a macro expansion 3.12, 3.21
Exit from a program 1.4, 2.16
ExNext 2.5
Expressions 3.6
External file, insertion 3.22
External main memory 3.1
External name, definition 3.22
External references 1.2, 3.22
External symbols 3.12, 3.21, 3.22, 4.1

FAIL 3.12, 3.18
FAILAT 1.4
Failure of routines 1.3
File handle, initial output 2.9
File handles, closing 1.3
File handles, opening 1.3
File handling calls 2.2, 2.19
File lock 2.8
File order (ALINK) 4.4
File position, find 2.11
File, close for I/O 1.4, 2.2
File, delete 2.3
File, examine 2.4, 2.5
File, find length of 2.11
File, open for I/O 2.8
File, read bytes from 2.9
File, rename 2.10
File, unlock 2.12
File, write bytes to 2.13
FileInfoBlock 2.4
Files, special metacc 1.7
Filing system locks 2.8
Find and point at logical position in file 2.11
Find current position in file 2.11
Find length of open file 2.11
Flag an error 3.18
Format 2.1
FORMAT 3.12, 3.18
Forward reference 3.14, 3.15
Free locks 1.4
FROM (ALINK) 4.2, 4.3
Functions 2.1

General directives 3.13, 3.22
Generate a user error 3.18
Generate an assembly error 3.12
Greenhills C compiler 1.5

Header (ALINK) 4.5
Header files 1.1, 3.2
Hexadecimal 3.7

I/O 1.3, 1.4, 2.2, 2.15
I/O handling 2.15
I/O redirection (> <) 1.3
I/O streams 1.4
I/O, closing files for 2.2
IBM PC serial port (AUX) 1.10
IDNT 3.13, 3.23
IFC 3.12, 3.19
IFD 3.12, 3.19
IFEQ 3.12, 3.19, 3.21
IFGE 3.12, 3.19
IFGT 3.12, 3.19
IFLE 3.12, 3.19
IFLT 3.12, 3.19
IFNC 3.12, 3.19
IFND 3.12, 3.19
IFNE 3.12, 3.19, 3.21
Immediate data 3.10
INCLUDE 3.13, 3.22
INCLUDE directive nesting 3.22
Include dirlist 3.2
Include files 1.1
Including 3.2
Including files 3.2
Indexed addressing 3.1
Info 2.6
Initial I/O file handles 1.3, 2.6, 2.9
Initialized data 3.13
Input 1.3, 2.6
Input file handle 2.6
Input to linker 4.1
Insert an external file 3.13, 3.22
Instruction destinations 3.10
Instruction names 3.4, 3.7
Instruction types 3.10
Instructions 3.1, 3.4, 3.7, 3.10
Interface, quick cross-reference 2.19
Internal format of date and time 2.15
Internal registers 3.1
IoErr() 1.3, 2.7
IsInteractive 2.7

JSR (Jump SubRoutine) 1.2
Jumps 3.1

Keyword template for ALINK 4.2
Keywords (ALINK) 4.2, 4.3

L Long Branch specifier 3.5
L Long Word-sized data (32-bits) 3.5
Label field 3.4, 3.14, 3.15
Labels 3.4, 3.11, 3.20, 3.22
Length of title 3.17

Level of a node 4.6
Libraries (Linker) 4.1
LIBRARY 4.2, 4.3, 4.4
Library base pointer 1.2
Library code segments 4.5
Line length 3.12, 3.17
Link map 4.1, 4.2, 4.3, 4.5, 4.8
Link program 1.2
Linker 1.1, 1.3, 1.5, 1.7, 1.10, 4.1-10
Linker output 4.5
LIST 3.12, 3.16, 3.22
Listing 3.12, 3.16
Listing control 3.12, 3.16
Listing file 3.2, 3.3
Listing, turn on/off 3.12, 3.16
LLEN 3.12, 3.17
Load code 1.2, 1.3, 1.4, 2.16, 2.20
Load file 1.7, 4.1, 4.2, 4.5
Load file destination 4.2
Load module into memory 2.17
Loader 1.2
Loading programs 1.2, 1.3, 1.4
LoadSeg 2.17
Local labels 3.3, 3.4
Lock 2.8
Lock, duplicate 2.4
Locks 1.4, 2.2, 2.3, 2.4, 2.5, 2.8, 2.12, 2.15
Locks, free 1.4
Logical AND (&) 3.6
Logical NOT (˜) 3.6
Logical NOT 3.6
Logical OR (!) 3.6
Logical position in file, find and point at 2.11
Long word alignment 2.4, 2.6, 3.18
Long word operations 3.1
Long word size 3.8, 3.15
Lower case, use of 2.1
Lshift (<<) 3.6

MACRO 3.12, 3.20
Macro definition, start 3.20
Macro definition, terminate 3.21
Macro directives 3.12, 3.20
Macro expansion 3.20, 3.21
Macro invocations 3.5
Macro, contents of a 3.20
Main memory 3.1
Makeamiga 1.9
Makefile 1.9
MAP (ALINK) 4.1, 4.2, 4.3, 4.5
MASK2 3.13, 3.22
Memory 1.4, 2.17, 3.1, 3.4
Memory address 3.4
Memory size 3.1
Memory variant 3.10
Memory, direct access of 3.1
Memory, load module into 2.17
Memory, return 1.4
Message destination (ALINK) 4.2
Metacc 1.5
MEXIT 3.12, 3.20, 3.21
Monadic minus (-) 3.6
Motorola specification, extensions to 3.4, 3.7
MS-DOS 1.5, 1.10
Multiple definitions of labels 3.4
Multiply 3.6

NARG 3.12, 3.20
Nesting macro calls 3.20
NEWCLI 2.17
NIL: 2.8
Node level 4.6, 4.7
Node ordinate value 4.6
Nodes of the overlay tree 4.5
NOFORMAT 3.12
NOL - see NOLIST
NOLIST 3.12, 3.16, 3.22
NOOBJ 3.12, 3.18
NOPAGE 3.12, 3.17
Null string 3.19, 3.23
Numbers 3.7

Object code 3.11, 3.12
Object file 1.5, 3.2, 3.3, 3.18, 4.1, 4.2
Object module 3.18
Octal 3.7
OFFSET 3.11, 3.14
Offset definition 3.11
Offset from alignment boundary 3.18
Offset termination 3.14
Opcode field 3.4, 3.5
Open 2.8
Open file for I/O 2.8
Open file, find length of 2.11
Operand field 3.5, 3.6, 3.14, 3.15, 3.16, 3.17, 3.19, 3.22
Operand types for operators 3.6
Operand word 3.1
Operands 3.6
Operation codes 3.5
Operation word 3.1
Operator precedence 3.6
Operators 3.6
Options to the assembler, passing 3.2
Order of overlay files 4.8
Ordinate value of a node 4.6, 4.7
Origin, absolute 3.11
Origin, relocatable - see RORG

Output 1.3, 2.9
Output file handle 2.9
Output, change 1.8
OVERLAY 4.3, 4.5, 4.6
Overlay file order 4.8
Overlay files 4.1, 4.4, 4.8
Overlay number 4.9
Overlay references 4.8, 4.9
Overlay supervisor 4.1, 4.5, 4.8, 4.9
Overlay symbols 4.9
Overlay terminator 4.6
Overlay tree 4.1
Overlaying 4.1, 4.5, 4.4, 4.5, 4.8, 4.9
Overriding operator precedence 3.6

PAGE 3.12, 3.16
Page heading 3.17
Page length 3.12, 3.17
Page throw 3.12, 3.16, 3.17
Parallel port 1.7, 1.8, 1.9
Parameter files 4.1, 4.2, 4.3
ParentDir 2.9
Permanent register value, assign - see EQUR
Permanent value, assign - see EQU or REG
PLEN 3.12, 3.17
Postincrement address 3.10
Precedence of operators 3.6
Primary input 4.1, 4.4
Priority, process 2.14
Process handling calls 2.14, 2.20
Process identifier 2.15
Process, create a new 2.14
Process, delay 2.15
Program assembly 1.2
Program compilation 1.2
Program counter (PC) 3.1, 3.4, 3.13, 3.14
Program encoding (Assembler) 3.3
Program end 3.11
Program entry 1.2
Program loading 1.2, 1.3
Program section 3.11, 3.13
Program source 3.22
Program termination 1.4
Program title 3.12, 3.17
Program transfer 1.2
Program, exit from 2.16
Program, link 1.2
Protection, set file or directory 2.12

R Relative symbols 3.6
RAM-based resident library 1.2
RAW: 2.8
READ 1.2, 1.3, 1.9, 1.10, 2.9
Read bytes from file 2.9
Reference external name 3.12
Reference to symbols 4.8
Refering to a standard set of definitions 3.22
REG 3.11, 3.15
Register 3.7
Register corruption 1.3
Register list, define 3.15
Register names 3.4, 3.7
Register values 2.1
Registers, address 3.1
Registers, data 3.1
Registers, internal 3.1
Relative expression 3.8
Relative origin, set 3.13
Relative symbol value 3.7
Relative symbols R 3.6
Relocatable code 3.13
Relocatable memory locations, assigning 3.13
Relocatable origin 3.11
Relocatable symbol 3.22
Relocation information, generation of 3.22
Rename 2.10
Repetitive linking 4.1
Reserve memory locations 3.16
Reserved symbol 3.20
Resident library 1.1, 1.2, 2.2
Results 2.1
Return allocated memory 1.4
ROOT (ALINK) 4.3
Root stack 2.14
RORG 3.11, 3.13, 3.14
Routines, failure of 1.3
Rshift (> >) 3.6
RTS 1.4
Running a program under Workbench 1.4

S option 3.3
S Short Branch specifier 3.5
Scanned library 4.2
SECTION 3.11, 3.13, 3.14
Seek 2.11
Segment identifier 2.18
Segment list 2.14
Segment, unload 2.18
Send 1.8
Separation character 3.2, 3.4, 3.5
Serial line 1.7, 1.9
SET 3.7, 3.11, 3.15
Set line length 3.12, 3.17
Set page length 3.12, 3.17
Set program title 3.12, 3.17

Set relative origin 3.13
Set symbol value 3.15
SetComment 2.11
SetProtection 2.12
Shared read lock 2.8
Size specifier 3.5, 3.8, 3.15, 3.16
Skip lines 3.12
Source code 3.20
Source files 1.5
Source statement syntax 3.4
Source termination 3.14
Sources of input to linker 4.1
SP 3.7
Space 3.2, 3.17, 3.20
Space at the start of a line 3.3
SPC 3.12, 3.17
Special metacc files 1.7
Special register 3.1
Special symbol (NARG) 3.12
SR 3.7, 3.14
Stack 1.3
Stack allocation 1.3
Stack checking 1.3
Stack pointer (SP) 1.4, 3.1
Stack size 1.3
Stack, size of root 2.14
Standard I/O 1.3
Standard macro definitions, include 3.22
Start a macro definition 3.20
Startup code 1.2, 1.3
Status register (SR) 3.1
Storage, define 3.11
String descriptor 3.22
Strings 3.6, 3.8, 3.19, 3.22
Subtract (-) 3.6
Sun, cross development under a 1.5, 1.7, 1.8, 1.9
Supervisor mode 3.1
Symbol cross reference table 4.1
Symbol defined, assemble if 3.19
Symbol definition 3.11, 3.14
Symbol dump 3.3
Symbol not defined, assemble if 3.19
Symbol reference 4.8
Symbol value, set 3.15
Symbols 3.6
Syntax 2.1
Syntax for address mode 3.9
System information, return 2.7
System stack pointer - see stack pointer

Temporary value assignment - see SET
Terminate a macro definition 3.21
Ticks 2.15
Time, internal format 2.15
Timeout character arrival 2.13
Title header 3.17
Title length 3.17
Title of program 3.12
Title of program, set 3.17
TO 4.2, 4.3
Transfer, program 1.2
Tree nodes 4.5
Tree specification 4.3
Tree structure 4.5, 4.8
TTL 3.12, 3.17

Uninitialized data 3.13
Unique label generation 3.20
Unix 3.2
Unload code 1.4
Unload segment 2.18
UnLoadSeg 2.18
UnLock 2.12
Unlock a directory 2.12
Unlock a file 2.12
Unnamed sections 3.13
Upper case, use of 2.1
User error 3.18
User mode 3.1
User stack pointer - see stack pointer
User symbol 3.4, 3.14
Using less memory 4.5
Using the Linker 4.2
USP 3.7, 3.14

Values 2.1
Variants 3.10
VER (ALINK) 4.2
Verification file 3.2
Virtual terminal, connection to a 2.7, 2.13
Volume 2.10

W Word-sized data (16-bits) 3.5
WaitForChar 2.13
Warning messages 3.2, 4.4
WIDTH 4.3
WITH 4.2, 4.3, 4.5
Word boundary, alignment on 3.15
Word operations 3.1
Word size 3.8, 3.15
Workbench 1.3, 1.4
Write 1.3, 2.13

X option 3.3
XDEF 3.12, 3.21
XREF 3.12, 3.22, 4.3, 4.5

Commodore Business Machines, Inc.
1200 Wilson Drive, West Chester, PA 19380

Commodore Business Machines, Limited
3370 Pharmacy Avenue, Agincourt, Ontario, M1W 2K4