ALN Linker

Introduction

The ALN linker takes object modules or libraries of object modules, created by an assembler or high-level language compiler, and links them together to form a single executable program file.

ALN can also link in binary files created by art tools, music tools, sound tools, and other such programs which create data files with information that has to be included in your program. By accepting these files directly, ALN can save you time and disk space.

The Command Line

Below is the basic format of the ALN command line:

```
aln [options] <input files>
```

ALN understands a wide variety of command line switches which affect its mode of operation. These are listed and described below.

For input files, ALN understands both Alcyon-format[1] and BSD-format[2] object files and object archive libraries. ALN can create either Alcyon-format or COFF encapsulated format executable files, either with or without symbols and debugging information.

Command Line Options

A summary of ALN's command line options is shown below. Note that all of these options **must** be specified before any of the input files are listed, with the exceptions of the -x, -i, and -ii options.

The ALN linker was originally distributed as part of the Atari ST computer developer's kit, and has been updated to support the requirements of the development system for the Atari Jaguar. As a result, some of ALN's original features and command line options are not really applicable to Jaguar programming. They are listed for completeness and noted where appropriate, but the description of these features will be minimal.

---

1 The Alcyon format is also known to some people as the DRI format. It is a common object file format used on the Atari computer, originally by the Alcyon C compiler and associated tools in the Atari Computer's Development Kit. It's a basic, but not overly flexible object module format.

2 The BSD format is a very commonly used format for object modules on a wide variety of systems, primarily UNIX and similarily oriented systems. It is a very flexible format that allows for a wide variety of linker patch-up information and debugging information.

Switch	Description
-?	Print ALN usage information.
-a *text, data, bss*	Output absolute executable file (.ABS or .COF). ***This is the recommended output option for Jaguar Programming.*** *text* = Address for TEXT segment *data* = Address for DATA segment *bss* = Address for BSS segment Values for *text*, *data*, and *bss* can be: a hexadecimal value to be used as the address. **r**: relocatable segment (not useful for Jaguar programs) **x**: contiguous segment (contiguous with previous segment) For example "-a 802000 x 4000" would put the TEXT segment at $802000, the DATA segment immediately after that, and the BSS section at $4000. By default, an Alcyon format executable will be created (*.ABS) unless the **-e** option is also used.
-b	Don't remove unreferenced local labels
-c *[name]*	Add contents of *[name]* to the command line. They are read and processed as though they appeared on the command line. Any command line switch may be used. Arguments in the file may be delimited by whitespace (tabs, spaces, newlines) or commas. As with the regular command line, only the **-i**, **-il**, or **-x** options may be used after the first input file is specified. This option is used to get around the system's limitation of 128 byte maximum command line length. It is typically the last [illegible]

Switch	Description
-i *fname label* **-ii** *fname label*	Includes the binary data contained in the file specified by *fname* in the link. The contents of the file are placed verbatim into the DATA section. ALN creates a global symbol named *label* with the value of the starting address and another global symbol name *labelx* with the value of the ending address+1. (e.g. if *label* is "picture" then you get a label named "***picture***" at the start and a second label named "***picturex***" at the end). Wih the **-i** option, the symbol created will be truncated to a maximum of 8 characters length. (The end symbol will be truncated to 7 labels before the 'x' is added, for a total of 8 characters.) With the **-ii** option, the symbol will not be truncated (assuming that you have specified COFF-format output). This option is used within the list of input files. It's similar to the MADMAC directive **.incbin**.
-k *symbol*	Adds *symbol* to the kill list
-l	Add local symbols to output file (as well as global symbols) This option is like a stronger version of the **-s** option.
-m	Produce load symbols map on standard output. The load map contains each symbol's name, value, and type. The load map lists only global symbols unless the **-l** option is used. The symbol types are encoded as follows: C: Common F: File G: Global A: Archive (only with "File") E: External Q: eQuated L: Local R: Register
-n	Output no file header to ABS file (output raw image of TEXT & DATA sections)
-o *fname*	Set output filename to *fname*. If *fname* has an extension (e.g. ".COF"), then that extension is used. Otherwise, a default extension is appended (".COF" for a COFF-format absolute executable, ".ABS" for an Alcyon format absolute executable, or ".PRG" for a GEMDOS-format relocatable executable). If the **-o** option is not specified, the output file name is taken from the first linked file on the command line (including archives specified with **-x** and data files specified with the **-i** or **-ii** options), plus the appropriate extension. Note that if this would make the output file name the same as the first input file (e.g. "**aln -p A1.O A2.O**" which would use "**A1.O**" as the output file name because we are only doing a partial link), ALN will abort: in this case, **-o** must be specified.
-p	Partial link. collect the named object modules and libraries together and create a single object module, suitable for later passes through ALN.
-q	Partial link with nailed-down BSS. This is the same as the **-p** option, except that all symbols in the COMMON section are resolved into the BSS section.

Switch	Description
-r[*size*]	Section alignment size. Automatically pad the size of each object module's TEXT, DATA, and BSS sections so that the size is an integral multiple of the specified size.. *size* is one of: w: word (2 bytes) l: long (4 bytes) p: phrase (8 bytes, default alignment) d: double phrase (16 bytes) q: quad phrase (32 bytes) For example, the option **-rp** would cause the TEXT. DATA. and BSS sections of each object module in the link to be padded in size until they were a multiple of 8 bytes.
-s	Generate a symbol table in the output file, and include all global symbols. Use the **-l** option (by itself) to include local symbols as well as globals.
-u	Don't abort on unresolved, externally defined symbols. The unresolved symbols are listead on standard output, but the link proceeds as if their

The input object modules would be START.O, KEYPAD.O, DRAW.O, INIT.O, VIDEO.O, SOUND.O, and OBJLIST.O. Also included would be the binary data file IMAGE.DAT, which would be referenced via the *img_data* label.

Unfortunately, the command line above would never work in real life because it is longer than 127 bytes. Both MSDOS and the Atari computer's GEMDOS operating systems have a maximum command line length of 127 bytes. To get around this, we need to have a linker command file that specifies some of the command line options and/or input files. Normally, you would specify your options in the first part of the command line and put the names of your input files into the linker command file. So we would probably really do something like this instead:

```
-u -w -v -v -a 802000 x 4000 -o showimg.cof -c
```

```
aln -e -f l -rp
showimg.lnk
```

mandline is the same, but then it ends with the **-c showimg.lnk** option instead be linked. This option tells ALN that there are more linker commands in the NK. This file would contain something like this:

```
 draw.o
objlist.o
_data
```

```
start.o keypad.o
video.o sound.o
-i image.dat img
```

e as long as required to specify all of your input files and options.

The command file can b

ite Library Path

Filenames and t

th object modules and archive libraries, in both the current default directory and as the *library path*. This is specified either by the ALNPATH environment ne command line using the "**-y**" option. If both the ALNPATH variable and e present, then the command line specification takes precedence.

ALN looks for files, bot in the directory named a variable, or named on th command line option ar

be a full pathname which names a single directory, like "E:\JAGUAR\LIB". uding drive letter, should be specified.

The library path should The complete path, incl

n a file, it looks in a number of places. First it tries to open the file exactly as directory. If that fails, ALN then appends a ".O" extension and tries again. If ks in the *library path* directory for the specified filename. If that still fails, then O" extension again looks in the *library path* directory again. If none of these N gives up. For example, if you specified "mathsubs" to include O", then ALN would look for:

When ALN tries to ope specified, in the current that fails, then ALN loo ALN then appends a ". methods work, then AL "E:\LIB\MATHSUBS.

Attempt	Filename searched for	Result
1	mathsubs	*fails*
2	mathsubs.o	*fails*
3	E:\LIB\mathsubs	*fails*
4	E:\LIB\mathsubs.o	*succeeds!*

Of course, as soon as a matching file is found, ALN stops looking. A filename can also contain a part
name. If you want to use the archive "E:\LIB\LOCAL\MYLIB" and your *library path* is "E:\LIB" then
listing "LOCAL\MYLIB" on the command line is sufficient. ALN will look for:

Attempt	Filename searched for	Result
1	LOCAL\MYLIB	*fails*
2	LOCAL\MYLIB.O	*fails*
3	E:\LIB\LOCAL\MYLIB	*succeeds!*

ALN never tries to append the ".O" **extension** to a filename that already has an extension. Also, ALN will not look in the *library path* for **filenames** that start with "\" or "/" or which contain a colon (:). The assumption is that such filenames are **based on** a specific drive or the root directory of the current drive, and therefore adding them to the library **path** specification would not work.

Absolute Linking

An absolute link is one for which the **-a** option is specified. ***This is the type of link normally used for Jaguar Development.*** Note that the **-a** option takes three arguments: the base address for the TEXT, DATA, and BSS segments, respectively. **The** base address can be specified in the following ways:

- A hexadecimal value, which is **taken as the** starting address of the segment.
- The letter 'r', which stands for "**relocatable**".
- The letter 'x', which stands for "**contiguous** with the previous segment" (whether that segment is absolute or relocatable).

During an absolute link, an absolute **object module** is produced, which includes the base address of each segment in its header. In Jaguar **development**, this file can be used directly with the debugger as an executable program file.[3] See the section **File Formats** for more details.

In an absolute object module, all references to an absolute segment have already been resolved; that is, there is no relocation information for them, because they are not relocatable. References to relocatable segments still have relocation information associated with them. If there are no references to relocatable segments (either because there are no **such segments**, or no references to them), the relocation information is missing entirely, and a flag in the header indicates this.

For example, when linking a program to be placed in ROM, ALN might be used to link with the TEXT and DATA segments contiguous, starting at the address of the ROM (say, $802000), and with the BSS segment at some address in RAM (say, $4000). This would be done with ALN as follows:

```
aln -o rom.abs -a 802000 x 4000 romfile.o
```

[3] This is typically the desired output for Jaguar programming.

Alternatively, a program with its data segment in ROM, but with relocatable text and BSS segments, could be linked as follows:

```
aln -o romdata.abs -a r 802000 r romfile.o
```

Of course, it would be up to the program loader to perform the TEXT and BSS relocation at execute time. and this does not really apply to Jaguar programming.

File Symbols

ALN will generate file symbols when the **-f** option is used. A file symbol appears at the start of each object module in the symbol table. Its name is the name of the module, its value is the start of the text segment of that module, and its type is TEXT FILE ($0280). With these symbols, you can determine which object module a given symbol came from, because the symbols from a module immediately follow its file symbol.

ALN also generates a file symbol at the start of each archive: this is a special symbol in that its name is the name of the archive, but its type is TEXT FILE ARCHIVE ($02C0). Furthermore, a second symbol is generated at the end of the archive: it has the same type, but its name is blank. This signals the end of the previous archive.

The use of bit 6 of the type field to mean "archive" is not an original part of the Alcyon symbol-table standard. As such, some older tools can not be expected to understand it.

File Formats

There are three basic types of files that ALN deals with: object modules, archive libraries (containing object modules), and executable program files.[4] There are two different styles of file format for each of these file types: Alcyon format and BSD/COFF format.

The different Alcyon formats originate with the Alcyon C compiler, an original component of the Atari Computer Development Kit dating back to 1985, and on other systems before that. We will discuss them first.

e program files have the same basic format: Header
age of Text segment (program code), image of Data
(debugging information), Relocation Information (used by
g program into memory).

Alcyon Format Files

Alcyon format object modules and executable
(information describing the file contents), im
segment (pre-initialized data), Symbol Table
linker during link and/or by OS when loading

-ii options as part of this list, because ALN doesn't really care what
y included verbatim into the DATA section of the output file.

4 We don't consider data files included via the -i or
the contents of such files might be; they are simpl

The header includes information such as the sizes of the other segments and the actual file type (encoded in a "magic" number). Any segment may be empty or missing except the header.

Alcyon-Format Object Modules

A standard Alcyon-format (relocatable) object module header has the following format:

```
struct oheader {
      int magic;          /* the magic number 0x601A */
      long tsize;         /* text segment size */
      long dsize;         /* data segment size */
      long bsize;         /* bss segment size */
      long ssize;         /* size of the symbol table *
      char reserved[10];  /* ten unused bytes (must be zero) */
};
```

All values are in Motorola (big-endian) format. Following the header is the module's text segment, the module's initialized data segment, the symbol table information, and then the module's relocation fixup information.

Alcyon-Format Relocatable Executable Program Files

Alcyon-format executable programs (.PRG files) have almost the same format as relocatable object modules. The header is the same (except that the *magic* field is $601B instead of $601A), and the text and data segments, plus the symbol table, follow. The overall file format could be defined in 'C' as:

```
struct oheader theHeader;
char text_segment[theHeader.tsize]
char data_segment[theHeader.dsize]
char symbol_table[theHeader.ssize]
char fixup_info[]; /* arbitrary size */
```

This type of file is not used in Jaguar programming. It is mentioned here because it is similar to the Alcyon flavor of the executable file format is typically used for Jaguar programming (described in the following section).

[MF1]

Alcyon-Format Absolute Object Modules (Jaguar Executable Program)

This file format is similar to the standard object module and relocatable executable file formats, except that there is normally no relocation information to allow the file to be loaded at any address. Instead, the address references in the code and data have been absolutely positioned by the linker. The file header has been expanded to specify the load address for the TEXT, DATA, and BSS segments. The absolute object module header has the following format:

```
struct abshdr {
     int magic;              /* the magic number 0x601B */
     long tsize;             /* text segment size */
     long dsize;             /* data segment size */
     long bsize;             /* bss segment size */
     long ssize;             /* size of the symbol table */
     long reserved;          /* an unused longword */
     long textbase;          /* the base of the text segment */
     int relocflag;          /* zero if reloc info exists */
     long database;          /* the base of the data segment */
     long bssbase;           /* the base of the bss segment */
} theHeader;
char text_segment[theHeader.tsize]
char data_segment[theHeader.dsize]
char symbol_table[theHeader.ssize]
```

Normally, a relocatable file uses a base address of $00000000 for all internal references, and relies on the system loader to use the relocation table to relocate the references as necessary to the address where the file's TEXT segment is loaded. In contrast, an absolute-linked file uses a base address for each segment that is defined at link time, and normally does not include relocation information. However, it is possible for an absolute file to contain relocation information.

If there is any relocation information, the *relocflag* field in the header will be zero, and that information will follow the symbol table (if any). If the *relocflag* field is not zero (and in particular if it is minus one), there is no relocation information. This is always the case when none of the three segments is relocatable, but it can also happen if there are no references to a relocatable segment (e.g. the text segment is relocatable, but contains position-independent code, and the data and BSS segments are absolute).

Alcyon-Format Archive Libraries

Archives are files containing other files, usually relocatable object modules. The "header" of an archive file is simply the magic number $FF65 (hex). The archived files consist of a header, then the object module file itself. The next file follows immediately. A zero word follows the last file in the archive. The archived-file header is as follows:

```
struct arheader {
      char a_fname[14];     /* the file name */
      long a_modti;         /* the last-modified time */
      char a_userid;        /* not used in TOS */
      char a_gid;           /* not used in TOS */
      int a_fimode;         /* the file's mode word */
      long a_fsize;         /* the file's size in bytes */
      int reserved;         /* zero */
};
```

The remainder of the archive file, which is *a_fsize* bytes in length, immediately follows the header.

BSD/COFF File Formats

BSD-Format Object Modules
COFF-Format Absolute Executable Program Files
BSD-Format Object Module Archive Libraries

Information on these file formats has not been reformatted into the main ALN documentation as yet. This information will be available in a future revision.

Duplicate Symbols In Modules

he symbol value
ported by
in the case of
last definition

ne first instance

ame symbol.

When the same symbol is exported (declared as global) from multiple object modules, t
exported from the first such module will take precedence. When the same symbol is ex
multiple modules in one archive, the last such module will take precedence. Therefore,
two archives exporting the same symbol (from modules exporting needed symbols), the
in the first archive is the one which will be used.

However, if an archive is included with **-x**, the modules are read in archive order, and th
of a symbol is the one which prevails.

Unless the **-w** flag is used, you will get no notification that multiple files exported the sa

Unused Modules In Libraries

e it to be out-of-
o a given link.

Since the dependency information is built from the archive, certain conditions can cause
date with respect

chive *Z* contains modules *M* and *N*, and *M* depends on *N* because it needs symbol *S*,
Z will reflect this. But if the symbol *S* is exported by a file *Y* earlier in a particular
N is not actually needed at all.

For example, if ar
the index file for *Z*
link, then module

dule *N* from the archive, but will then notice that both *N* and *Y* are exporting symbol
ce a warning message if the **-w** option is specified. Finally, since *Y* occurs earlier in
e value of symbol *S* is taken from *Y*. ALN will notice that module *N* is not in fact
process, and will discard *N* completely, with another warning message.

ALN will read mo
S. This will produ
the link than *N*, th
used in the linking

gos

Error Messag

on error messages from ALN are self-explanatory; for instance, "**File <x> is not an**
cases, however, a little more explanation is in order.

Most of the comm
archive." In some

ing an external reference in the
ngle-word. This can happen if

Some errors refer to a 16-bit fixup overflow. This means that in resolv
file, a value greater than 32767 or less than -32768 had to be put in a si

you have a PC-relative reference to a symbol more than 32K away. This is only a warning, since you might be using the value as an unsigned integer (in which case it might not be an overflow).

Other errors report that they occurred at a given offset (always hex) in a given module. The offset is always in bytes, counting from the beginning of the text segment of that module.

If the solution to eliminating the source of an error is giving you difficulty, please contact Jaguar Developer Support for assistance.

DOINDEX -- Alcyon-Format Archives And Their Indexes

ALN requires that an index file exist for each Alcyon-format archive library which is included in a link (but not BSD-format archive libraries). This index file has the same name as the archive, with the extension ".NDX", and should be in the same disk directory as the archive itself. If ALN can not find an index file for an archive you name, it will produce an error message to that effect and abort.

The DOINDEX utility builds an index file for the named archive (regardless of whether one already exists). If desired, DOINDEX will also print a human-readable index of the archive on standard output, and inform you of symbols which are declared global in more than one module in the archive. The last such declaration is the one which will prevail when that archive is used in the linking process.

The command line options for DOINDEX are as follows:

Option	Description
-i	Index: print an index of the archive to the standard output, including the name of each module, the global symbols it exports, and the external symbols it imports. Finally, list the symbols which are external to the archive (imported by modules in the archive but not exported by any of them).
-w	Warnings: produce warnings about duplicate symbols in the archive.

The last argument to doindex is the name of an archive. Doindex opens that archive, builds its index file, and writes that file to *file*.ndx in the same disk directory as *file* itself.

The index file contains dependency information so the linker does not have to go through the whole archive to resolve all the symbols. It consists of information about each module in the archive, the name of each symbol exported by any module in the archive and the module which exports it, and a dependency list for each module, stating, "if you need module A, you will also need modules B, C, and D." During linking, this information is collected together for each symbol which is unresolved at the time the archive appears in the command line, and only the needed modules are read in from the archive.