MIT Concurrent VLSI Architecture Memo 62

Massachusetts Institute of Technology
Artificial Intelligence Laboratory

Guide to the M-Machine Linker

Yevgeny Gurevich

Abstract

This memo outlines the usage and features of version 1.2.1 of the M-Machine linker, **lnk**, and the format of the M-Machine object files and executables. Specific issues discussed include creation and maintenance text and data segments, symbol tables, relocation information for linking, and the linking process. A discussion of debugging tools is also included.

Keywords: M-Machine, linker, executable.

1 Overview

This document details the usage and features of the M-Machine Linker, `lnk`, which links multiple M-Machine object files into files suitable for execution on the M-Machine simulator, `msim`. Included is an overview of the object file format, executable file format, and a discussion of symbol-table and relocation use and maintenance. Finally, in addition to its use as a linking tool, `lnk` may also be used for a variety of debugging purposes. These debugging features as discussed as well.

2 Lnk Usage

`lnk` may be invoked with no arguments to see a usage summary. To link a set of object files to into an executable, list the files on the command line:

```
lnk file_1.out file_2.out ... file_n.out
```

An optional `-o<outfile>` argument may be used to denote an output executable name. If no output file is specified, the resulting executable is called `a.out`.

The lnk executable has an exit value of 1 if an error occured, and 0 otherwise.

3 Lnk Compatibility

Lnk is compatible with version 2.1 and higher of the M-Machine Assembler. It currently recognizes both regular object files, and debugged object files (corresponding to files compiled with the assembler's debug option.) Debug object files contain relocation information and a tagged text segment which is used in showing object-file disassembly.

4 M-Machine object file format

There are currently two The M-Machine object file formats. Both formats closely follow the standard for UNIX object files as specified in the `a.out.h` and `exec.h` standard C header files. Specifically, each object file contains a 32-byte header which contains information on the file's segments, and relocation and symbol tables. Table 1 summarizes header-file contents.

The contents of the object file header may be used to access the file's individual components. Standard macros may be used to find specific file offsets. Macros in the UNIX standard `a.out.h` such as `N_TXTOFF`, `N_DATOFF`, etc. may be used.

4.1 Text and Data Segment

The object file text segment begins at byte 33 of the object file (if the header is considered to be in bytes 1-32) and continues for the number of bytes specified in the text segment size field of the file header. Addresses in the text relocation table of the object file are referenced off the beginning of this local text segment, so address 64 in a relocation entry denotes the operation which starts at byte 97 of the object file.

Debugged object files contain a tag byte for every 64-bit word in the text segment which denotes whether the following 64-bits are operations, or a 64-bit code data word. Therefore, the true size of the text segment (one with the tag bytes stripped) in bytes is 8/9 of that denoted in the file header.

Bytes	Description
1	Object-file version information. The 7-bit value starting at bit one of this byte contains the object file type. Currently, 0x1 denotes a regular object file, while 0x41 denotes a debugged object file.
2	Machine type (unused.)
3-4	Magic Number (currently, the magic number of the platform where lnk was compiled.)
5-8	Text Segment Size. The size of the object file's text segment in bytes.
9-12	Data Segment Size. The size of the object file's data segment in bytes.
13-16	BSS Segment Size. The size of the object file's bss segment in bytes.
17-20	Symbol Table Size. The size of the object file's symbol table.
21-24	Execution Entry. The byte address in the text segment where program execution should start.
25-28	Text Relocation Size. The size of the text relocation table in the object file.
29-32	Data Relocation Size. The size of the data relocation table in the object file.

Table 1: M-Machine object file header spec

The data segment begins where the text segment ends and continues for the number of bytes specified in the header data segment size field. Data reloction addresses reference off the beginning of this data segment, so address 64 denotes the data word staring at byte (97 + size of the text segment in bytes).

4.2 Text and Data relocation entries

Text and data relocation entries exist for each symbol which was referenced in the original massl file. Each relocation entry is a 12-byte structure of the form `AAAAIIITEEEE` where `AAAA` denotes the address into the text or data segment of the operation or data word which referenced the symbol. `T` identifies the type of relocation entry. If the msb is set, the relocation is for an external symbol reference. The 5 lsb's determine whether the exact type of relocation (summarized in table 2.)

Value	Relocation Type
0	absolute 7-bit class A, C, E, or I operation symbol reference.
1	absolute 11-bit class B operation symbol reference.
2	absolute 18-bit class G operation symbol reference.
3	relative 18-bit class G operation symbol reference.
9	64-bit integer symbol referenced in a data word.
11	64-bit floating-point symbol referenced in a data word.
12	64-bit M-Machine pointer symbol referenced in a data word.

Table 2: relocation types

`III` is either the index into the local symbol table of the symbol referenced or a generic symbol type, depending on the symbol type. For debugged-type object files, this usually denotes a symbol-table index.

Due to the fact that debugged object files contain relocation entries for each data word (text segment code data words the data segment words), the special value of `0xffffff` denotes a relocation entry which has no corresponding symbol in the symbol table. `EEEE` is currently unused.

4.3 Symbol Table

The symbol table is appended to the object file data relocation table and contains entries for each symbol which is referenced by relocation entries. Each symbol-table entry is of the form `IIIITUUUVVVV`

`IIII` contains a 4-byte index into the object file string table (appended to the symbol table) where the null-terminated ascii-string naming the symbol is stored. `T` is the type of symbol (local or global. text, data, or absolute, defined or not.) `U` is unused. `VVVV` contains the 32-bit value of the symbol. For this reason, no symbols in current M-Machine object files may reference addresses longer than 32 bits.

The string table following the symbol table begins with a 4-byte integer denoting the size of the string table in bytes. Therefore, the minimum size of a string table is 4 bytes in the case that it contains no strings. Null-terminates strings in the string-table are stored one after another, and their starting characters are referenced by the symbol-table index fields.

5 Linking Object Files

Lnk proceeds through several phases when creating an executable file from a list of object files. Linking is only meaningful if more than one object file is specified to be linked. In the first phase, all object files are opened, and have their relevant sections (text segment, data segment, text relocation, data relocation, string table, and symbol table) read into data structures. If any errors are encountered while filling data structures, the object files are most likely corrupt or of the wrong format. In this case, lnk aborts with an error message. If all files are successfully read, lnk examines the symbol tables of input object files and looks for multiply-defined external symbols, or references to locally-undefined external symbols for which no definitions exist.

If an external symbol is defined in more than one module, lnk records this and then aborts with a list of all multply-defined external symbols. Similarly, if an external symbol is referenced but never defined in any module, it is recorded and a list of such undefined symbols is displayed. In either case, lnk aborts the linking phase.

If the symbol-tables of input object files are in order, each undefined symbol entry in a module's symbol-table is patched with the appropriate globally-visible value.

The next phase of linking involves patching text and data segments with new values. *All references to text and data symbols need to be patched*, unless those symbols are local absolute symbols. Patching involves the use of relocation entry address fields to find text and data addresses which need to be patched, and the symbol-index fields to determine which symbol-table entries to use when looking for replacement values.

Because text and data symbols have their values changed, operations which once referenced absolute text symbols may not be unable to do so during the patch phase since new values might not fit into immediate slots. The linker prints error diagnostics if it is unable to place a symbol value into the appropriate operation.

Once patching is complete, text segments are concatenated and written to the output executable file. Data segments are similarly concatenated and written out. Text and data relocation tables are created for the linked executable with references to a new executable symbol table (constructed from individual symbol tables of the multiple modules.) The relocation, symbol-table, and string-table are then written to the executable. Finally, the executable's header is modified to reflect correct byte-sizes of all segment, relocation, and symbol information. If the global symbol `Main` is found within the executable's symbol table, the `a_entry` field of the header is filled with the symbol's value. This value represents the abolute byte offset into the text segment where execution should begin. If no such symbol is found, the value `0x0` is used.

6 Debugging Features

The debugging features of lnk allow display and disassembly of debugged object or executable files. Files which are assembled with the M-Machine assembler's debugging option are debugged object files. Lnk can disassemble the text and data segments, display a listing of relocation information and symbols which are referenced by each file, and show the local symbol table.

6.1 Text Segment Disassembly

The text segment diassembly prints out a listing of the entire file's packed text segment, with embedded code data, if any. Each line of the disassembly listing is of the form

```
$AAAAAAAA: [XXXXXXXX] disassembled instruction
```

where `AAAAAAAA` is the operation's address in the file's text segment, `XXXXXXXX` is the 32-bit encoding of the operation as it appears in the object file, and a pretty-printed disassembly follows. Nops may be supressed from appearing in the disassembly.

If code data appears embedded in the text segment, it appears in similar form, except that that `XXXXXXXX` is replaced with `DATA_TYPE` where the type is either `INT`, `PTR`, or `FP`. The disassembly is then the value of that data item. Pointers are printed as pointers, floats are printed as 64-bit ieee floating point numbers, and integers are printed as is.

6.2 Data Segment Disassembly

Unsorted The data segment is displayed as a list of ascending addresses and their contents:

```
$AAAAAAAA: <TYPE> <VALUE>
```

where `AAAAAAAA` is the address of `VALUE` (which is a `TYPE` data type) in the data segment.

Sorted The sorted data segment option uses an iterator function to display entries in data segments sorted by type. Current types recognized and supported are 64-bit integer, 64-bit IEEE floating point, and 64-bit M-Machine pointer.

Table 3 summarizes debugging command line options.

7 Bug Reports and Comments

Please send all bug reports, comments, or questions regarding the linker to `yev@ai`.

Option	Description
-Di	Disassembles the text segment of each file given on the command line.
-Dis	Disassembles the text segment and skips nops.
-Dd	Displays the data segment of each file on the command line.
-Du	Displays the data segment of each file on the command line, sorted by data type.
-r	Displays the text and data relocation entries for each file on the command line.
-s	Displays the local symbol table of each file on the command line.
-h	Displays the file header.

Table 3: lnk debugging command line options