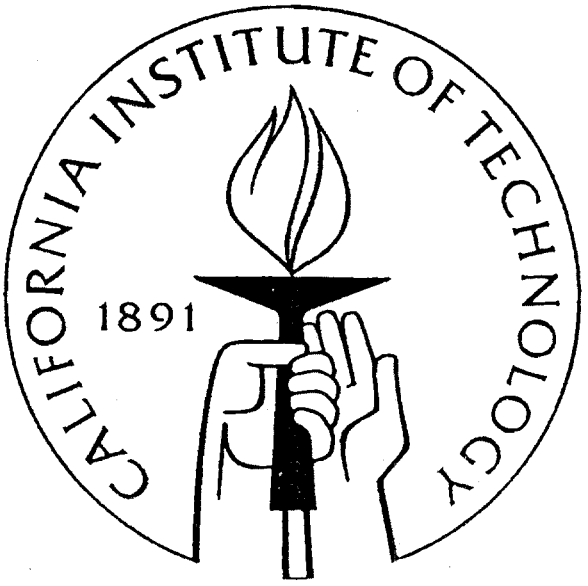

System Tools for the J-Machine

Daniel Maskit
Yair Zadik
Stephen Taylor

Computer Science Department
California Institute of Technology

Caltech-CS-TR-93-12

System Tools for the J-Machine [1]

Daniel Maskit, Yair Zadik and Stephen Taylor
Scalable Concurrent Programming Laboratory
California Institute of Technology

April 30, 1993

1 Introduction

This document contains a description of the system tools we have developed to support programming the J-Machine. The intent of this document is to provide sufficient details to allow other programming systems to utilize the tools. This document is a technical reference for the use of implementors; it is not intended for casual readers.

The basic programming model supported by the tools provides the abstraction of a fully connected set of n multicomputer nodes numbered 0 to $n-1$. Only sequential code execution and remote function invocation are supported at each node. Global variables are replicated at every node.

Executable code consists of a collection of functions which may be distributed onto nodes arbitrarily. The underlying runtime system ensures that code is transported to each node as needed. To enhance system performance, functions are separated into two groups: those that are distributed across the entire machine and those that are replicated at every node. Distributed functions are intended to be used infrequently. Replicated functions are frequently-used code segments such as floating point routines, the microkernel, and critical application functions. The placement of distributed functions is based on a bin-packing module within the linker. This is accessed using a standard interface to allow the user to experiment with different code allocation schemes. Code distribution conserves limited memory without excessive speed degradation.

The tools provided include the GNU C compiler retargetted for the J-Machine, an assembler, linker/archiver, and loader. The linker maps a program onto a physical address space. This address space resides upon the virtual nodes of the abstraction. The loader

[1]The research described in this report is sponsored primarily by the Advanced Research Projects Agency, ARPA Order number 8176, and monitored by the Office of Navel Research under contract number N00014-91-J-1986. The first author is partially supported by an NSF Graduate Research Fellowship.

then maps the abstraction's virtual node-space onto the machine's physical node-space. The programs that make up the tool suite are:

- mdc: The GNU-based message-driven C compiler.
- mdas: The Caltech version of the MIT assembler for the J-Machine.
- ar: The standard BSD UNIX archiver.
- mdranlib: The Caltech version of ranlib.
- mdld: The Caltech linker.
- mdload: The Caltech loader.

In addition to these tools, we have defined a set of file formats that provide the interface between various tools. A standard set of interface functions are defined for manipulating these formats.

2 Interface Function Definitions

This section contains function prototypes for the complete set of interface definitions and brief descriptions of all of the supplied functions. These allow reading and writing object, executable and library files. Details on the data structures contained within the files are supplied in Appendix A. The differences between our library format and the standard UNIX file format are described in Appendix B. Examples of the use of the library are provided in Appendix C.

2.1 File Opening and Closing

The following routines open and close object, library, and executable files for use with other routines. The mode corresponds to the same mode in fopen and may be "r", "r+", or "w" for executables and "r", "r+", or "a+" for libraries. NULL or zero is returned if an error occurs.

```
JOBJ *jobj_open(char *name, char *mode);
JLIB *jlib_open(char *name, char *mode);
JEXE *jexe_open(char *name, char *mode);
int jobj_close(JOBJ *);
int jlib_close(JLIB *);
int jexe_close(JEXE *);
```

2.2 Symbol Table Manipulation

The symbol table is contained in object, library and executable files. The following functions start reading symbols from the beginning of this table.

```
void jobj_startsym(JOBJ *);
void jexe_startsym(JEXE *);
```

Read the next symbol from an executable or object file. The symbol references are returned as an array in refp.

```
SYMBOLENTRY *jobj_nextsym(JOBJ *, SYMBOLREF **refp);
SYMBOLENTRY *jexe_nextsym(JEXE *, SYMBOLREF **refp);
```

Change the fixed portion of the entry for the symbol just read. Changing the refcount field will corrupt the file. This routine can only be used in "r+" mode and is intended to allow changes to various flags in the the symbol table entries.

```
int jobj_changeprevsym(JOBJ *, SYMBOLENTRY *sym);
int jexe_changeprevsym(JEXE *, SYMBOLENTRY *sym);
```

Write a symbol and its references to a file. The references are either an array of offsets or a linked list of (count, array of offset) pairs (a REFLIST).

```
int jobj_writesym(JOBJ *, SYMBOLENTRY *sym, SYMBOLREF *ref);
int jexe_writesym(JEXE *, SYMBOLENTRY *sym, SYMBOLREF *ref);
int jobj_writesym_reflist(JOBJ *, SYMBOLENTRY *sym, REFLIST *refl);
int jexe_writesym_reflist(JEXE *, SYMBOLENTRY *sym, REFLIST *refl);
```

2.3 Text and Data Section Manipulation

Read the text or data section of the file into a malloc'd buffer and returns the buffer with its length in length.

```
char *jobj_text(JOBJ *, int *length);
char *jobj_data(JOBJ *, int *length);
char *jexe_data(JEXE *, int *length);
```

Write the text or data to the end of the section of the file. Returns zero if an error occurs. Writes to different sections may not be intermixed, i.e. no writedata, writename, or writesym, may occur between any two writetext of the same file.

```
int jobj_writetext(JOBJ *jobj, char *buf, int len);
int jobj_writedata(JOBJ *jobj, char *buf, int len);
int jexe_writedata(JEXE *jobj, char *buf, int len);
```

Return the current offset within the text or data section being written. The next buffer

written will be placed at this offset.

int jobj_textpos(JOBJ *)
int jexe_textpos(JEXE *)
int jobj_datapos(JOBJ *)
int jexe_datapos(JEXE *)

Create an empty node information table for a file which will be distributed over n nodes. Returns null if can not allocate that much memory.

NODEINFO *jexe_create_nodeinfo(n)

Write a function whose code is in the buffer to the text section and updates the node information table nodes (which was allocated using jexe_create_nodeinfo). node is the node number being updated or SEC_REPL for replicated functions. No write to node text or other section may occur between two writes to the same node's text section.

int jexe_writenode(JEXE *, char *buf, int len,
NODEINFO *nodes, int node, int funcid);

Write the node info table in buf to the file. buf should be allocated using jexe_create_nodeinfo().

int jexe_writenodeinfo(JEXE *, NODEINFO *buf, int num);

2.4 Name Section Manipulation

Return a null terminated string which is the name table entry pointed to by namep.

char *jobj_name(JOBJ *, int namep);
char *jexe_name(JEXE*, int namep);
char *jlib_name(JLIB *, int namep);

Write the name to the end of the name table and returns a namep for it or -1 if an error occurs. The same restrictions on intermixed writes applies. jlib_writename should only be used in "a+" mode.

int jobj_writename(JOBJ *, char *name);
int jexe_writename(JEXE *, char *name);
int jlib_writename(JLIB *, char *name);

2.5 Header Information Manipulation Routines

Executable files contain extra information which is useful for communication between the linker and the loader. This information is not present in object files. The number of nodes the executable is distributed over, the origin of the data section, the length of zero

intialized data, the function id of the entry point of the executable, and the origin of the kernel table are returned by these functions.

```
int jexe_num_nodes(JEXE *);
int jexe_datao(JEXE *);
int jexe_zerol(JEXE *);
int jexe_entryf(JEXE *);
int jexe_kernelo(JEXE *);
```

Note that the entry point is always formatted as a distributed function id. If a replicated function is the entry point, then it is treated as a distributed function at node 0 (since it really can be called at any node).

This information can be set with the cooresponding jexe_set_ routines:

```
int jexe_set_datao(JEXE *, unsigned value);
int jexe_set_zerol(JEXE *, unsigned value);
int jexe_set_entryf(JEXE *, unsigned value);
int jexe_set_kernelo(JEXE *, unsigned value);
```

The number of nodes the executable is distributed over is set while writing the node information table using jexe_writenodeinfo.

2.6 Library Manipulation Routines

Return the offset within the archive of the object file containing the global symbol with the specified name. Zero (an invalid offset) is returned if name is not found.

```
int jlib_offsetfor(JLIB *, char *name);
```

Open the object file at at_offset for reading. This file should not be closed with jobj_close. It is automatically closed by the next jlib_jobj_at or jlib_next_jobj, or by jlib_close on the library. NULL is returned if an error occurs or if the offset is the offset of the index entry. The ar header for the file is in jlib→ar.

```
JOBJ *jlib_jobj_at(JLIB *, int at_offset);
```

Open the next object file in the archive for reading and return the offset of its ar header. It is automatically closed by the next jlib_jobj_at or jlib_next_jobj, or by jlib_close on the library. After opening the archive, the next file is defined to be the first file in the archive. If an error occurs or the file is a library index, the position in the archive is not changed and NULL is returned. offset_returned is zero if no more files exist in the archive. The ar header for the file is in jlib→ar.

```
JOBJ *jlib_next_jobj(JLIB *, int *offset_returned);
```

Write the library index. Since this index must be sorted, this call should only be made

once per archive. This routine should only be used in "a+" mode.

int jlib_writeent(JLIB *, int n, LIBENTRY *le_array);

Delete the file at the current position from the archive. This can be used to delete an old index entry when reading with jlib_next_jobj. Zero is return if an error occurs.

int jlib_delentry(JLIB *);

3 mdas: Assembler

This section outlines the functionality and output requirements for the new MDP assembler. The enhanced assembler provides support for some new features. The intent of the output requirements is to provide necessary support for the archiver and linker. The syntax of the assembler has been enhanced. Only the additions are described in this document. Please see the MIT documentation [2] for a description of the original assembly language syntax. The rationale for rewriting the assembler, and creating additional tools is to support:

- Relocatable code
- Multiple input files
- Code distribution and partitioning

These goals were achieved through:

- Late binding of symbols.
- Definition of a standard object file format.
- Creation of an indexable binary format. Use of a symbol table in the object file. Definition of a program as a set of functions, rather than a monolith.

3.1 Changes to Assembly Language

This section summarizes improvements to the assembly language syntax which facilitate code distribution and separately compilable modules; it also describes some minor changes from MDPSim (the MIT assembler for the J Machine) which simplify compiler code generation.

1. It is essential that the assembler distinguish between functions and global variables. MDPSim had been altered to allow for the usage of a :: label delimiter (i.e. _foo::). This delimiter was treated identically to :. In the new assembler, :: is treated as a function entry point. : is used as a delimiter for all other labels.

2. Support for sections (data, text) is implicit in the object file format. The assembler must support this paradigm via TEXT and DATA directives in order to generate the proper output. The linker will rearrange sections so arithmetic on labels in different sections is not supported.

3. Support for definition of data blocks. That is, rather than having 50 lines each of which contains 'DC INT:0' to clear out a 50 word block, we allow for a UBLOCK 50 directive. This generates a data symbol entry with the appropriate symbolic name, marked as uninitialized. This allows for smaller executables and shorter download times. It also distinguishes uninitialized data from data explicitly initialized to 0 so that "common" data is supported.

4. Support for linker size consistency checks. In order to allow the linker to catch different definitions for the same symbol, there needs to be size information for all data symbols. Uninitialized data uses the form specified in (3) above. Initialized data takes the form:

   IBLOCK n
   n lines each of which defines one word of data

5. Support for assembler directives specifying scope of data and functions. Data and functions which are known to be defined within a module and are global in scope are specified with:

   GLOBAL *symbol_name*

   Functions which are referenced within a module, but may be defined elsewhere are specified with:

   FUNCTION *function_name*

   Symbols which are referenced within a module, but may be defined elsewhere are specified with:

   EXTERNAL *symbol_name size*

   For both FUNCTION and EXTERNAL, it is legal for a definition of the symbol (denoted by GLOBAL) to occur elsewhere in the same module.

6. Support for an assembler directive denoting the end of a function:

   ENDFUNC

This allows the assembler to distinguish between the local data associated with one function, and the code of an adjacent function. The linker may rearrange functions, so arithmetic on labels in different functions is not supported.

3.2 Support for Late Binding

Because some symbols referenced in a given module are defined elsewhere, and have no value during assembly, their resolution must be deferred until link-time when it is possible to take a global view of the whole program. In a distributed program, due to the necessity of runtime symbol binding to support a global name space, this class of symbols is larger. In order to provide maximum flexibility at link time, the new assembler outputs:

- Entry points for all functions, including symbolic names.
- Storage requirements, initialization values and names for all global data.
- Location of all references to functions, global variables and undefined symbols.

3.3 Object File Format

The object file format has been designed to supply the same functionality as standard UNIX file formats such as a.out and COFF. The key points of this design are:

- Flexibility of format
- Separation between data and text to allow remapping of global data separately from code relocation.
- Information on all references to global symbols to simplify resolution as symbols are defined.
- Support for arithmetic operations on symbols to permit complex symbolic addresses.
- Support for system tool version tracking.
- Easy manipulation of values in text and data sections.
- A single format for both executable and object files.

To implement these ideas we use:

- A file header which indicates start and length of all sections within the binary file.

- Entries within the header for both text and data sections.
- Detailed symbol table entries including lists of references.
- Allowance for multiple symbols referenced from one location. Definition of constant assignment defined for the assembler. Description of a linker paradigm for performing the necessary calculations.
- Data and text sections consist of 36-bit MDP words formatted as a pair of 32-bit words representing tag and value fields.
- Header size information and reserved fields to extend the format if needed.
- Version numbers for system tools tracking
- Storage within the file header for a Magic Number to easily distinguish object and executable files from each other and standard Unix files.
- Additional fields for use in executables in both the header and symbol table.

As long as the header is the first item to appear in a file, the ordering of all other sections is arbitrary and left to the implementer. The offsets specified in the header are byte offsets measured from the beginning of the file. All lengths are in bytes unless otherwise stated. This allows for objects and executables to be read from within an archive file. The sections of the file are:

- **Text section** – Contains code including some constant data.
- **Data section** – Contains global, static, and constant data.
- **Symbol table** – Contains naming, size, type and reference information for all global symbols in the file and most local symbols.
- **Name table** – Contains ASCII, newline-terminated strings representing names of symbols.

The layout of the text, data and name sections are simply described within preceding comments. The symbol table is discussed in §3.3.1.

3.3.1 The Symbol Table

The purpose of the symbol table is to provide:

- Enough information to navigate from one table entry to the next.

- Access to the ASCII name of the symbol.
- Sizing information for symbols to allow for consistency checking.
- Symbol type information (code or data).
- Addressing information for symbol.
- Information on references to the symbol.

Once a symbol has been resolved, the list of reference locations is processed. For each location the value of the symbol is added to the contents of the reference. Complex arithmetic is supported by allowing multiple references for one location. For example, take the address:

$$\_FOO + (8 \times \_BAR) + 9$$

The assembler will initially output the value of the constant part of this expression:

$1:$000000009

The location will be listed in the symbol table as having one reference to _FOO, and 8 references to _BAR.

When _FOO is resolved (let us say to the value $1:$00001400), this reference becomes:

$$\$1 : \$000000009 + \$1 : \$00001400 = \$1 : \$00001409$$

When _BAR is resolved (let us say to the value $1:$00010000), we end up performing 8 additions, one for each reference. This yields:

$$\$1 : \$000001409 + \$1 : \$00010000 = \$1 : \$00011409$$

$$\$1 : \$000011409 + \$1 : \$00010000 = \$1 : \$00021409$$

$$\vdots$$

$$\$1 : \$000071409 + \$1 : \$00010000 = \$1 : \$00081409$$

4 ar: Archiver

The system toolkit includes the use of the standard UNIX archiver **ar** for building the archive and **mdranlib** which builds a symbol table and inserts it into the archive in a manner analogous to the standard BSD UNIX utility **ranlib**. The library file format is that defined as the output of **ar**. The details of the symbol table for the archive are described in Appendix B.

5 mdld: Linker

The linker provides the essential functionality that exists in a typical UNIX linker such as **ld**. In addition it performs the tasks necessary to support distributed-code management. These tasks are:

- Creation of a symbol table describing all functions within an executable file for use by the loader
- Generation of Unique Function Identifiers that provide a global name space for functions.
- Replacement of function references with IDs for distributed code management
- Remapping of global variables to support one-copy-per-node paradigm

The rest of this section describes the processing necessary to tie together functions with each other, and with global data. By default, all functions are distributed. There are a variety of command-line switches to override this default. These switches are described in the linker manual pages.

5.1 Function Handling

In the standard UNIX linker the connection between functions is made by making the addresses of functions available to their callers. In the distributed-code model the only useful 'address' for a function is its ID, which consists of a *home node* number and a physical address at that node. Accordingly, the J-Machine linker needs to generate these IDs, and make them available where they are needed.

Prior to generating the function IDs, the linker performs bin-packing to equalize the amount of code that is resident at each node. Replicated code is treated much like data in that space is allocated for it at every node. Space for each distributed function is allocated only at its home node. As discussed in [1], the compiler plants code to request microkernel intervention for locating functions. Distributed functions must be referenced through this scheme; replicated functions are generally accessed directly by address at runtime by patching out the call to the microkernel.

It should be noted that this mechanism makes address arithmetic using function references meaningless. Also, the bin-packing and ID scheme require that the size of the machine be specified at link time. The resulting machine code can not be properly loaded and executed on a machine with fewer nodes, although it may be executed on a larger machine.

5.2 Global Data Handling

At link time, a contiguous block of all global variables in a program is created. One copy of this block exists at each node during runtime. All references to global variables are recomputed to use the addresses within this block. There is no attempt to provide global data coherence between nodes.

5.3 Executable File Format

It is desirable to have a single format that can support both the needs of both the linker and loader so that:

- Fully linked executables may be relinked for different environments
- Partial linking may be supported (currently not implemented)
- One set of routines may be used for implementing the file format

Thus, the layout of an executable file is identical to an object file, but its contents differ in:

- Different magic numbers
- File header fields for address of start of data, length of zeroed data, ID of first function, and the start of a special kernel table
- An additional section, the node table, exists containing a table that divides the text section into two subsections for replicated code and one subsection per node for distributed code
- The symbol table contains no undefined symbols, only contains functions, and the functions are group grouped by home node number with all replicated function first

The node table supplies information on the location and length of the text for each node within the executable and on its address within physical memory on that node.

6 mdload: Loader

The loader reads executables generated by the linker, allocates the machine, maps virtual nodes onto the physical machine, downloads the appropriate code and data to each node, and executes the program.

The virtual machine used in the node numbering scheme by the linker is a useful abstraction that, unfortunately, is not supported directly by the hardware. Before the program can be executed, linear virtual machine node numbers must be translated into the J-machines three dimensional relative node numbers. Since such a translation can not be done efficiently at runtime, this operation is done at load time by patching the node number in all references to distributed functions based on a list produced by the linker in the output file.

After the patching process, each node is sent all initialized data, all replicated code, and the distributed code appropriate to that node. Additionally, a small table supplying the microkernel with some necessary information about the program (entry point, first unused location, etc.) is sent to each node. The microkernel is responsible for zeroing the zero-initialized data section before executing user code.

While the program is executing, the loader handles I/O between the executing program and the host.

References

[1] Maskit, D. and Taylor, S., "A Message-Driven Programming System for Fine-Grain Multicomputers", Submitted for publication, March 1993.

[2] Noakes, M., "MDP Programmer's Manual", MIT Concurrent VLSI Architecture Memo 40

A Data Definitions for File Formats

Not all defined fields are used in the both the object and executable binary formats. Fields marked "JEXE only" are not used in the assembler output. They are for executables only.

File header:

```
typedef struct {
    int magic;          /* magic number - file type is JEXE or JOBJ */
    int vers;           /* version number of file format * 100 */
    int hdrl;           /* size of header in words */
    int textp;          /* offset of start of text section */
    int textl;          /* length of text section */
    int datao;          /* origin of data section (JEXE only) */
    int datap;          /* offset of start of data section */
    int datal;          /* length of data section */
    int zerol;          /* length of zeroed (uninit) data section (JEXE only) */
    int symp;           /* offset of start symbol table */
    int syml;           /* length of symbol table */
    int namep;          /* offset of start of name table */
    int namel;          /* length of name table */
    int entryf;         /* function number of first function to invoke (JEXE only) */
    int nodesp;         /* offset of start of node section table, (JEXE only) */
    int nodesl;         /* length of node section table (JEXE only) */
    int kernelo;        /* start of kernel function table (JEXE only) */
    int reserved[8];    /* decrement size of this when adding fields */
} FILEHEADER;
```

The first three fields provide for identification of the file format and version. Except for zerol, fields whose name ends in "p" or "l" provide information on the layout of the file. Sections within the file may be in arbitrary order. The rest of the fields are for information only and are used by the loader and kernel. The last field is a few words reserved for future use.

Node Table:

```
typedef struct {
    int p;          /* offset to start of text for node */
    int l;          /* length of text for node */
    int o;          /* origin of text for node as funcid */
} NODEINFO;
```

There is one such entry per node. Additionally, there are two such entries for replicated functions which preceed the entries for each node. Only the second of these entries is

currently used. The first is reserved for replicated functions in on-chip memory, a feature not yet supported by the microkernel. Entries should be indexed by taking the section name (SEC_ON_CHIP, SEC_REPL) or node number and adding it to FIXED_SECS.

Symbol Table:

```
typedef struct {
    int entryl;      /* Length of entry */
    int namep;       /* Offset in bytes of name in name table */
    int symsize;     /* Size in words of symbol */
    int symtype;     /* symbol type (SYMT_*) flags */
    int symorg;      /* Offset of symbol (UNDEFADR when undefined) */
    int refcount;    /* Number of references to this symbol */
    int entry;       /* Offset of function entry from symorg or UNDEFADR */
    int funcid;      /* Function ID (JEXE only) */
} SYMBOLHEADER;
```

The symbol table is composed of a list of contiguous entries, each of which consists of a fixed-length SYMBOLHEADER, and a possibly zero-length list of locations of references to the symbol. Each reference location is an offset in machine words into either the data or text section of the file. The entry field is ignored if the symbol is not for a function.

A.1 Notes on Symbol Table

The **symtype** field is actually a set of flags that tell the linker some information about the symbol. The following predefined constants can be used to set the appropriate flags by OR'ing the appropriate values together:

- **SYMT_FUNC** – symbol is function in text section
- **SYMT_VAR** – symbol is variable in data section
- **SYMT_LOCAL** – symbol's scope is local to file
- **SYMT_GLOBAL** – symbol's scope is global to program
- **SYMT_INIT** – symbol is a initialized symbol definition
- **SYMT_UNINIT** – symbol is uninitialized or undefined
- **SYMT_DIST** – if SYMT_FUNC, distributed function
- **SYMT_REPL** – if SYMT_FUNC, replicated function

- **SYMT_ON_CHIP** – if SYMT_FUNC, on chip replicated function
- **SYMT_MARKED** – symbol's SYMT_DIST/REPL/ON_CHIP set explicitly

The last four flags are to communicate information to the linker using the **jmark** utility. Macros beginning with the prefix IS_ (IS_FUNC, IS_REPL, etc.) can be used to test if the various flags are set. The following predefined constants (when inverted) can be used as mask out groups of related flags:

- **SYMT_M_SEC** – mask for symbol section (func or var)
- **SYMT_M_SCOPE** – mask for scope of symbol
- **SYMT_M_INIT** – mask for init/uninitialize flag
- **SYMT_M_LOC** – mask for function location (dist, repl, on-chip)

The function id output by the linker is should be accessed using the following macros:

- **ALL_NODES** – special constant indicating function is present on all nodes
- **FUNC_NODE(sym)** – node number of the function in SYMBOLENTRY sym
- **FUNC_ADR(sym)** – start address of the function in SYMBOLENTRY sym
- **FUNC_ID(n,adr)** – function id coorespond to a function at address adr on node n (or ALL_NODES)
- **DFUNC_NODE(id), RFUNC_NODE(id)** – node number from a function id for a distributed, replicated function
- **DFUNC_ADR(id), RFUNC_ADR(id)** – start address from a function id for a distributed, replicated function

Caltech loader addtionally transforms the virtual node number into a kernel node number (an NNR that is 8x8x64 as opposed to 32x32x64) using the macros:

- **NODE_NUM(x,y,z)** – kernel node number from x, y, z coordinates
- **NODEX(n), NODEY(n), NODEZ(n)** – x/y/z coordinate from a kernel node number

When outputting an undefined symbol reference, the assembler should place an INT:0 (1:0) in the text or data section as appropriate. If the symbol is to be added to a constant, it could be output as INT:constant (1:constant).

Bit 31 in references to symbols or the **symorg** field is cleared if location is offset into text section, set if location is offset into data section. Additionally, the special value UNDEFADR indicates an undefined address. Use the macros TEXTREF(offset), DATAREF(offset), and UNDEFADR when assigning to a reference. Use the macros IS_TEXTREF(ref), IS_DATAREF(ref), IS_UNDEF(ref), and REF_OFFSET(ref), when looking at the value of **symorg** or symbol references.

B Library File Format

The library file format is identical to the standard Unix "ar" file format with the following restrictions:

- all files, except for the last, must be valid object files
- the last file must be named "__.JLIB-INDX._" and contain a library index

The library index is divided into three sections: a name table, a table of index entries, and a trailer. The trailer appears as the last part of the index so it will always be at the end of the archive file and is thus easy to find.

Library Trailer:

```
typedef struct {
    int magic;      /* magic number to confirm really JLIB-INDX */
    int arhdrp;     /* offset of ar file header for index */
    int indxp;      /* offset of library index */
    int indxl;      /* length of index */
    int namep;      /* offset of name table */
    int namel;      /* length of name table */
}LIBTRAILER;
```

The magic number and the ar file header permit easy identification of the trailer as valid. The remainder of the entries are for locating and reading the other sections of the index.

The name table is composed of null terminate strings and is similar to the same section in object and executable files.

Library Index Entry:

```
typedef struct {
    int namep;      /* offset of symbol name within name table */
    int offset;     /* offset of ar file header for object file containing symbol definition */
}LIBENTRY;
```

The index entries are sorted by symbol name. Each entry consists of a symbol name and a file within the archive. The symbol name must be the name of a global variable or function defined within the file.

C Examples of Reading and Writing Binary Files

The following examples are intended to enable users to replace one or more of the tools in the tool set. The example of reading an executable file will be useful for replacing the loader. Those users intending to replace the assembler or to produce direct binary output from a compiler will find the example of writing an object file useful. Replacement of the linker can be facilitated by referring to both examples.

C.1 Reading Binary Files

This example, except where noted, is given in terms of reading an executable file. Reading object files is similar except the corresponding "jobj_" prefixed function is used instead of the "jexe_" function. Object files have one unified text section instead of a set of sections for the distributed text at each node and a section for the replicated text; therefore the text of an object is read differently than the text of an executable. Additionally, the contents of the symbol table are different: executables use all fields in the symbol structure, and do not contain symbols for data; object files contain symbols for both data and functions, and some fields in the symbol structure are unused. The difference in usage of fields in the symbol structure does not affect the read algorithm.

The following is an outline of the algorithm used by the loader to read an executable file:

1. Open the file.
2. Read the data section.
3. Read all text sections:

   Read the replicated text section.

   For each node, read the distributed text section for that node.
4. Optionally, read the symbol table and patch code and data as appropriate.
5. Read some miscellaneous information
6. Manipulate or download data, replicated code, and distributed code as appropriate.
7. Free any memory allocated for buffers by the read routines and close the file.

The sections may be read in any order. However, calls to read the symbol table (which unlike the other sections, can not be read with a single call) can not be interleaved with calls to read the other sections.

The following code fragments describe each of these steps in more detail.

```
/* 1. open file for reading */
JOBJ *fp;

if (!fp = jexe_open(pathname, "r")) {
    if (errno) /* If system failed to open file... */
        perror(pathname);
    else /* If file was not a J-machine executable */
        fprintf(stderr, "%s: not a J-machine executable file\n", pathname);
    abort();
}

/* 2. read the data section */
data_buffer = (j_word *)jexe_data(fp, &data_length);
```

For executable files, we have to read out a replicated section, and a section for each node over which the program is to be distributed. Information about length, origin (as a function id), and offset within the text section are also read at the same time.

```
/* 3a. read the replicate text */
repl_buffer = (j_word *)jexe_node(fp, SEC_REPL, &repl_length, &repl_funcid, &repl_text_offset);

/* 3b. read the distributed text at each node */
for (i = 0; i < jexe_num_nodes(fp); i++) {
    text_buffer[i] = (j_word *)jexe_node(fp, i, &text_length[i], &text_funcid[i], &text_offset[i]);
}
```

For object files, text is read the same way as data:

```
text_buffer = (j_word *)jobj_text(fp, &text_length);
```

For both object and executable files, read out the symbol and reference information.

```
/* 4. read the symbol table */ jexe_startsym(fp);
while (sym = jexe_nextsym(fp, &refs)) {
    /* process symbol sym and references refs */
    add symbol sym to symbol table;
    for (i = 0; i < sym→refcount; i++)
        add reference refs[i] to reference list for symbol sym;
    free(sym);
    free(refs);
}

/* 5. read the miscellaneous information from header */
```

```
address_of_start_of_initialized_data = jexe_dato(fp);
number_of_words_in_zero_initialized_data_section = jexe_zerol(fp);
distributed_function_id_of_entry_point = jexe_entryl(fp);
address_of_start_of_kernel_table = jexe_kernelo(fp);
```

At this point, all information necessary to manipulate the executable is available. For example, the current version of the Caltech loader would at this point map each virtual node number into a physical NNR, translate the function ID for each function into the form the current kernel needs based on this mapping, and use the list of references to patch each location referencing a distributed function so that it contains the translated function ID. This would not be necessary if the mapping of virtual node numbers to NNR were done on the processing node. Following this, the code is downloaded: data and replicated functions are broadcast to all nodes, while each distributed section is sent to the home node for that section. A small table containing the start of zero initialized data (calculated from start of data and length of data), the length of zero initialized data, the address of the kernel table, and the entry point of the program is also downloaded. The kernel clears the zero initialized data section and runs the program from the entry point given.

The read routines malloc space for the data that they return. At the conclusion of processing, these buffers should be freed.

```
/* 7. free buffers allocated by reads */
free(data_buffer);
free(repl_buffer);
for (i = 0; i < jexe_num_nodes(fp); i++)
    free(text_buffer[i]);

/* close the file */
jexe_close(fp);
```

C.2 Writing Binary Files

This example, except where noted, is given in terms of writing an object file. Writing executable files is similar except the corresponding "jexe_" prefixed function is used instead of the "jobj_" function. Object files have one unified text section instead of a set of sections for the distributed text at each node and a section for the replicated text; therefore the text of an object is written differently than the text of an executable. Additionally, the contents of the symbol table are different: executables use all fields in the symbol structure, and do not contain symbols for data; object files contain symbols for both data and functions, and some fields in the symbol structure are unused. The difference in usage of fields in the symbol structure does not affect the write algorithm.

The following is an outline of the algorithm used to write object files:

1. Open the file.
2. Write the text section.
3. Write the data section.
4. Write the name section.
5. Write the symbol table.
6. Close the file

In theory, the sections may be written in any order. However, in practice, the symbol table must be written out last since the value of some fields (specifically, the namep field, and depending on the application, the symorg field) in the symbol table entries can not be determined until the other sections have been written. Calls to write a section may not be interleaved with calls to write other sections in the same file.

```
/* 1. Open file for writing */
if (!(out = jobj_open(filename, "w"))) {
    perror(filename);
    abort();
}

/* 2. write out data section */
for (p = datasecs; p; p = p→next)
    if (p→len)
        /* For each non-zero length buffer, write out the buffer */
        if (!jobj_writedata(out, p→data, p→len)) {
            perror(filename);
            jobj_close(out);
            abort();
        }
```

For object files, the text section is written out the same way as the data section:

```
/* 3. write out object file text section */
for (p = textsecs; p; p = p→next)
    if (p→len)
        /* For each non-zero length buffer, write out the buffer */
        if (!jobj_writetext(out, p→text, p→len)) {
```

```
        perror(filename);
        jobj_close(out);
        abort();
    }
```

For executable files, this algorithm is a bit more complicated and is described at the end of this section.

```
/* 4. write out name table and remember the location of each name */
for (sym = first symbol; more symbols; sym = next symbol) {
    if ((sym→namep = jobj_writename(out, sym→name)) == (unsigned)-1) {
        perror(filename);
        jobj_close(out);
        abort();
    }
}
```

Notice that while writing the name table, we save the value returned from *jobj_writename* into the symbol table entry.

```
/* 5. write out symbols */
for (sym = first symbol; more symbols; next symbol) {
    int t;

    /* not shown: iterate over references, fixing them so they point
        to offsets in output file */
    if (!jobj_writesym_reflist(out, sym→sym, sym→refs)) {
        perror(filename);
        jobj_close(out);
        abort();
    }
}
/* 6. close the file */
jobj_close(out);
```

The algorithm presented above must be altered for executables. In executable files, there is a section for each node over which the program is to be distributed, plus one section for replicated functions. There is a table that contains information on each of these sections. This table should be dynamically allocated and cleared using *jexe_create_nodefinfo*. After all of the sections have been written out the node information table is also written out.

To facilitate the Caltech code distribution mechanism, each function is preceded in the executable by a word containing its length. The insertion of these length words complicates the writing out of node sections, and is omitted from the example code.

```
/* 3. Write text section in EXECUTABLE file */
nodes = jexe_create_nodeinfo(number_of_nodes);

/* this code assumes that the functions are grouped by node and sorted by address */
for (sym = first function; sym; sym = next function) {
    if (IS_DIST(sym)) /* If this is a distributed function... */
        /* Put it in the section for its home node */
        sec = FUNC_NODE(sym→funcid);
    else /* Otherwise, it is a replicated function */
        sec = SEC_REPL;

    /* Save the origin within the text section of the function we are about to write out */
    sym→symorg = jexe_textpos(out)/(sizeof (j_word));

    buffer = pointer to code for current function
    /* Write the function. */
    if (!jexe_writenode(out, buffer, sizeof(j_word)*sym→symsize, nodes, sec, sym→funcid)) {
            perror(filename);
            jexe_close(out);
            abort();
        }
}

/* write out the node information table */
jexe_writenodeinfo(out, nodes, number_of_nodes);
free(nodes); /* free node information table now that we're done with it */
```

Before closing an executable file, there are some additional fields in the header that need to be filled in. Object files do not require these fields. All of the other entries in the header are filled in by the library routines and should not be altered by user code.

```
/* Fill in the header fields used only in executables */
jexe_set_datao(out, origin of data section);
jexe_set_zerol(out, size in words of zero initialize data);
jexe_set_entryf(out, function id of entry point);
jexe_set_kernelo(out, origin of kernel table);

/* flush out buffers, write header information, and close the file */
jexe_close(out);
```