```
#ifndef DATASTRUCT_INCLUDED
#include "datastruct.h"
#endif

/* Data structure functions */
/* relocation */
RelocNode* RelocNode_create() {
    RelocNode *rnode = NULL;
    if ((rnode = (RelocNode*)malloc(sizeof(RelocNode))) == NULL)
      return NULL;
    rnode->next = NULL;
    rnode->entry = NULL;
    return rnode;
}

RelocNode* RelocEntry2RelocNode(Relocation_Entry *e) {
    RelocNode *rnode = NULL;
    if ((rnode = RelocNode_create()) != NULL)
      rnode->entry = e;
    return rnode;
}

void RelocList_add(RelocList *rlist, RelocNode *rnode) {
    if (rlist->head == NULL) {
        rlist->head = rnode;
        rlist->current = rnode;
    } else {
        rlist->current->next = rnode;
        rlist->current = rnode;
    }
    rlist->num_entries++;
}

/* symbol-table */
SymNode *SymNode_create() {
    SymNode *snode = NULL;
    if ((snode = (SymNode*)calloc(sizeof(SymNode),1)) == NULL)
      return NULL;
    snode->next = NULL;
    snode->entry = NULL;
    return snode;
}

void SymList_add(SymList *slist, SymNode *snode) {
    if (slist->head == NULL) {
        slist->head = snode;
        slist->current = snode;
    } else {
        slist->current->next = snode;
        slist->current = snode;
    }
    slist->num_entries++;
}

SymNode *FindSymTabEntry(SymList S, unsigned long index) {
    SymNode *snode = NULL;
    /* find the indexth entry in SymList... */
    /* if (S.num_entries == 0) return NULL; */
    for (snode = S.head; snode != NULL, index > 0;
         snode = ((snode == NULL) ? NULL : snode->next), index--) ;

    if (index || (snode == NULL)) return NULL;
    else return snode;
}

/* FileNode/List */
FListNode* FListNode_create() {
    FListNode* FLNode;
    if ((FLNode = (FListNode*)calloc(1, sizeof(FListNode))) == NULL)
      return NULL;
    FLNode->next          = NULL;
    FLNode->inFile        = NULL;

    FLNode->relocation.num_entries = 0;
    FLNode->datareloc.num_entries = 0;
    FLNode->symbol_table.num_entries = 0;
    FLNode->num_segments = 0;
    FLNode->segment_names = NULL;
    FLNode->segment_offsets = NULL;
    FLNode->DataSegment = NULL;
    FLNode->TextSegment = NULL;
    return FLNode;
}

FListNode* Filename2FileNode(char *filename) {
    FListNode* FLNode;
    if ((FLNode = FListNode_create()) != NULL)
      FLNode->inFileName = filename;
    return FLNode;
}

void inFileList_add(inFileList* InputFiles, FListNode *FLNode) {
    if (InputFiles->head == NULL) {
        InputFiles->head = FLNode;
        InputFiles->current = FLNode;
    } else {
        InputFiles->current->next = FLNode;
        InputFiles->current = FLNode;
    }
}

unsigned long TotalSymbols(inFileList FList) {
    unsigned long count = 0;
    FListNode *FLNode;

    for (FLNode = FList.head; FLNode != NULL; FLNode = FLNode->next)
      count += FLNode->symbol_table.num_entries;
    return count;
}

void CreateCombinedSymbolTable(inFileList* FList) {
    unsigned long a_syms = 0, i;
    SymNode *snode;
    FListNode *FLNode;
    char **sym_names;
    char duplicate_external = FALSE;

    /* allocate an array of symbols-name strings */
    sym_names = (char**)calloc(TotalSymbols(*FList), sizeof(char*));

    /* must try to get rid of duplicate global symbols */
    for (FLNode = FList->head; FLNode != NULL; FLNode = FLNode->next) {
        for (snode = FLNode->symbol_table.head;
             snode != NULL;
             snode = snode->next) {
            for (i = 0, duplicate_external = FALSE; i < a_syms; i++)
              if (!strcmp(sym_names[i], snode->entry->n_un.n_name) &&
                  (isExternalSymbol(*(snode->entry)) ||
                   isSegmentSymbol(*(snode->entry))) ) {
                  /* found an existing symbol already in the symbol table */
```

```
                  duplicate_external = TRUE;
                  break;
              }

            if ((duplicate_external == FALSE)
                /* && (snode->entry->n_type != 0xc0) */ ) {
                sym_names[a_syms] = snode->entry->n_un.n_name;
                a_syms++;
            }
        }
    }
    if (FList) {
        FList->combined_syms = sym_names;
        FList->combined_sym_count = a_syms;
    }
}

unsigned long FindSymbolInCombined(char *name, char **slist, unsigned long i) {
    unsigned long j;
    for (j = 0; j < i; j++)
      if (slist[j] != NULL)
        if (!strcmp(name, slist[j]))
          return j;
    return i;
}

void PrintCombinedSymbolTable(char **combined_symbols, unsigned long totsyms) {
    unsigned long i;
    for (i = 0; i < totsyms; i++)
      fprintf(stdout,"{%s}\n", combined_symbols[i]);
}

void FreeCombinedSymbolTable(char **slist, unsigned long i) {
    unsigned long j;
    for (j = 0; j < i; j++)
      if (slist[j] != NULL)
        free(slist[j]);
}

int isUndefSymbol(NList nl) {
    return ((nl.n_type - ((nl.n_type & N_EXT) ? N_EXT : 0)) == N_UNDF);
}
int isTextSymbol(NList nl) {
    return ((nl.n_type - ((nl.n_type & N_EXT) ? N_EXT : 0)) == N_TEXT);
}

int isDataSymbol(NList nl) {
    return ((nl.n_type - ((nl.n_type & N_EXT) ? N_EXT : 0)) == N_DATA);
}

int isAbsSymbol(NList nl) {
    return ((nl.n_type - ((nl.n_type & N_EXT) ? N_EXT : 0)) == N_ABS);
}

int isExternalSymbol(NList nl) {
    return (nl.n_type & N_EXT);
}

int isSegmentSymbol(NList nl) {
    return (nl.n_type == N_SEGMENT);
}
```

```c
/* M-Machine linker
   Yevgeny Gurevich, MIT AI Lab - Concurrent VLSI Architecture Group
   Spring, 1994 */

/* Linker data structures and accessor methods */

#define DATASTRUCT_INCLUDED

#ifndef LONG64
#define LONG64
#endif

#include <stdio.h>
#include <stdlib.h>
#include <a.out.h>

#include <mm_opcodes.h>

#ifndef LNKDEFS_INCLUDED
#include "lnkdefs.h"
#endif

typedef struct nlist NList;

typedef struct {
    unsigned long int r_address;     /* relocation addr (offset in segment)*/
    unsigned int      r_index   :24; /* segment index or symbol index      */
    unsigned int      r_extern  : 1; /* if F, r_index==SEG#; if T, SYM idx */
    int                         : 2; /* <unused>                           */
    unsigned int      r_type    : 5; /* type of relocation to perform      */
    long int          r_addend;      /* addend for relocation value        */
} Relocation_Entry;

typedef struct reloc_node {
    Relocation_Entry *entry;
    struct reloc_node *next;
} RelocNode;

typedef struct reloc_list {
    RelocNode *head;
    RelocNode *current;
    unsigned long num_entries;
} RelocList;

typedef struct sym_node {
    NList *entry;
    struct sym_node *next;
} SymNode;

typedef struct sym_list {
    SymNode *head;
    SymNode *current;
    unsigned long num_entries;
} SymList;

typedef struct flist_node {
    char *inFileName;
    FILE *inFile;
    unsigned char isDebug, isLinked;

    struct exec header;

    int            num_segments;
    char**         segment_names;
    unsigned long* segment_offsets;

    codebits      *TextSegment;
    char          *DataSegment;
    unsigned char *TextInfo;
    RelocList     relocation;
    RelocList     datareloc;
    SymList       symbol_table;
    struct flist_node *next;
} FListNode;

typedef struct inflist {
    char**    combined_syms;

    int       num_segments;
    char**    segment_names;
    int*      segment_offsets;

    unsigned  long combined_sym_count;
    FListNode *head;
    FListNode *current;
} inFileList;

RelocNode* RelocNode_create();
RelocNode* RelocEntry2RelocNode(Relocation_Entry*);
void       RelocList_add(RelocList*, RelocNode*);

SymNode*   SymNode_create();
void       SymList_add(SymList*, SymNode*);
SymNode*   FindSymTabEntry(SymList,unsigned long);

FListNode* FListNode_create();
void       inFileList_add(inFileList*, FListNode*);

unsigned long TotalSymbols(inFileList);
void          CreateCombinedSymbolTable(inFileList*);
void          FreeCombinedSymbolTable(char**,unsigned long);
void          PrintCombinedSymbolTable(char**, unsigned long);
unsigned long FindSymbolInCombined(char*,char**,unsigned long);

int         isUndefSymbol(NList);
int         isTextSymbol(NList);
int         isDataSymbol(NList);
int         isAbsSymbol(NList);
int         isExternalSymbol(NList);
int         isSegmentSymbol(NList);

/* FListNode*    CreateFileNode();
   int           AddFileNode(inFileList*, char*);
   void          AddSymNode(SymList*,SymNode*); */
```

```
/* M-Machine linker
   Yevgeny Gurevich, MIT AI Lab - Concurrent VLSI Architecture Group
   Spring, 1994 */

/* Main linker code */

/* #ifndef LNK_INCLUDED
   #include "lnk.h"
   #endif
   */

#ifndef LNKDEFS_INCLUDED
#include "lnkdefs.h"
#endif

#ifndef DATASTRUCT_INCLUDED
#include "datastruct.h"
#endif

#ifndef OBJECT_READ_INCLUDED
#include "object_read.h"
#endif

#ifndef OBJECT_UNPARSE_INCLUDED
#include "object_unparse.h"
#endif

#ifndef OBJECT_WRITE_INCLUDED
#include "object_write.h"
#endif

#ifndef PATCH_INCLUDED
#include "patch.h"
#endif

#ifdef __MSDOS__
#include <stdlib.h>
#else
#include <sys/stat.h>
int unlink(char *);
int fclose(FILE *);
int fflush(FILE *);
int fprintf(FILE *, char *, ...);
int fseek(FILE*, long int, int);
int fwrite(void *, int, int, FILE*);
#if 0
void free();
char *malloc(), *calloc();
int exit();
#endif
#endif

#define VERSION "1.2.20A"
#define DATE    "04-14-94"

typedef struct Options {
    inFileList InputFiles;          /* list of .out files.                 */
    unsigned char showHeader;       /* display input file header           */
    unsigned char showRelocation;   /* display input relocation information */
    unsigned char showSymTab;       /* display input symbol table          */
    unsigned char showInstructions; /* display disassembled instructions    */
    unsigned char debug;            /* flag: enable debugged output         */
    unsigned char skip_nops;        /* flag: skip nops when disassembling   */
    unsigned char show_data_sorted; /* flag: shows data when disassembling  */
    unsigned char show_data_unsorted; /*     shows data when disassembling  */
    unsigned char verbose;          /* flag: verbose linker output          */
    char *outputFileName;           /* name of output file                  */
    FILE *outputFile;
} Options;

static Options switches;

/* modifies: switches
 * effects:  Sets the default values for the lnk options fields. */
static void InitDefaultOptions() {

    switches.InputFiles.head = switches.InputFiles.current = NULL;
    switches.InputFiles.combined_syms = NULL;

    switches.debug              = TRUE;  /* for now, always output
                                            debugged object files */
    switches.showHeader         = FALSE;
    switches.showRelocation     = FALSE;
    switches.showSymTab         = FALSE;
    switches.showInstructions   = FALSE;
    switches.skip_nops          = FALSE;
    switches.show_data_unsorted = FALSE;
    switches.show_data_sorted   = FALSE;
    switches.verbose            = FALSE;
    switches.outputFileName     = NULL;
    switches.outputFile         = NULL;
}

static void PrintAboutInfo(FILE *target) {
    fprintf(target, "lnk %s %s by Yevgeny Gurevich, "
            "M-Machine software project\n",
            VERSION, DATE);
    fprintf(target, "                        "
            "Massachusetts Institute of Technology.\n");
}

static void PrintHelpText(FILE *target, char **text) {
    char **textPointer;
    for (textPointer = text; *textPointer; textPointer++)
      fprintf(target,"%s\n",*textPointer);
}

static void PrintMainUsage(FILE *target) {
    static char *help_message[] = {
        "",
        "Usage:\tlnk file_1 file_2 ... file_n -o<output>  -- link file_1..file_n to",
        "\t                                            form executable output",
        "\tlnk file_1 file_2 ... file_n [FLAGS]",
        "\t    FLAGS:     -h     -- Display file Header information",
        "\t               -r     -- Show Text and Data segment relocation",
        "\t               -s     -- Show symbol tables",
        "\t               -v     -- Verbose mode",
        "\t               -Di[s] -- Show instruction Disassembly (-Dsi skips nops)",
        "\t               -Dd    -- show unsorted data segment",
        "\t               -Du    -- show data segment sorted by data type",
        "",
        NULL };
    PrintHelpText(target,help_message);
    PrintAboutInfo(target);
}

/* modifies: argv, argc
 * effects: parses commandString and sets program flags
 * Modifies argv and argc to read further command strings
 * if necessary.
```

```
 */
static char **ParseRegularCommand(char *commandString, char **argv, int *argc)
{
    if (commandString == NULL) return argv;

    switch (*commandString) {
    case 'o':
        if (strlen(commandString) > 1)
          switches.outputFileName = commandString + 1;
        else {
            if (*argc && (argv+1)[0] && (*(argv+1)[0] != '-')) {
                switches.outputFileName = (++argv)[0];
                *argc = *argc - 1;
            } else {
                fprintf(stderr,"lnk: output file improperly specified.\n");
                exit(2);
            }
        }
        break;
    case 'D':
        if (strchr(commandString+1,'s'))
          switches.skip_nops = TRUE;
        if (strchr(commandString+1,'d'))
          switches.show_data_unsorted = TRUE;
        if (strchr(commandString+1,'u'))
          switches.show_data_sorted = TRUE;
        if (strchr(commandString+1,'i'))
          switches.showInstructions = TRUE;
        break;
    case 'h':
        switches.showHeader = TRUE;
        break;
    case 'r':
        switches.showRelocation = TRUE;
        break;
    case 's':
        switches.showSymTab = TRUE;
        break;
    case 'v':
        switches.verbose = TRUE;
        break;
    default:
        PrintMainUsage(stdout);
        exit(1);
    }
    return argv;
}

/* effects: closes all potentially open files
   Provide funtionality for cleanup before exiting the program. */
void CleanupLnk() {
    FListNode *FLNode;

    for (FLNode = switches.InputFiles.head;
         FLNode != NULL;
         FLNode = FLNode->next)
      fclose(FLNode->inFile);
    fclose(switches.outputFile);
}

int main(int argc, char *argv[]) {
    unsigned long textoffset, dataoffset;
    char *s, files_read = 0;
    FListNode *FLNode;
    SymList   undefList, duplist;
    struct exec outHeader;

    /* set up a standard set of options */
    InitDefaultOptions();

    /* parsing the command line arguments: continue traversing all of
     * the arguments, look for a preceding "-" in the arguments and, if
     * found determine which switch it was by looking at the next
     * character. */
    if (!(argc - 1)) {
        PrintMainUsage(stdout);
        exit(1);
    }

    while ((--argc > 0) && ((s = (*++argv)) != NULL)) {
        if (*s == '-')
          argv = ParseRegularCommand(argv[0] + 1, argv, &argc);
        else {
            if ((FLNode = FListNode_create()) != NULL) {
                FLNode->inFileName = argv[0];
                inFileList_add(&(switches.InputFiles), FLNode);
            } else {
                fprintf(stderr,"lnk: error allocating file node.\n");
                CleanupLnk();
                exit(1);
            }
        }
    }

    for (FLNode = switches.InputFiles.head;
         FLNode != NULL;
         FLNode = FLNode->next) {
        if ((FLNode->inFile = (FILE*)fopen(FLNode->inFileName,"rb")) == NULL)
          fprintf(stderr,"lnk: could not open %s\n", FLNode->inFileName);
        else {
            if (!ReadObjectFile(FLNode)) {
                fprintf(stderr,"lnk: Error reading object file information "
                        "for %s.\nlnk exiting...\n",
                        FLNode->inFileName);
                CleanupLnk();
                exit(1);
            }

            files_read++;

            if (switches.showHeader) {
                fprintf(stdout,"%s header:\n",FLNode->inFileName);
                DisplayHeader(stdout,FLNode->header);
                fprintf(stdout,"\n");
                fflush(stdout);
            }
            if (switches.showRelocation) {
                fprintf(stdout,"%s Relocation:\n",FLNode->inFileName);
                DisplayRelocation(stdout, FLNode->relocation,
                                  FLNode->symbol_table);
                fprintf(stdout,"\n");
                DisplayRelocation(stdout,FLNode->datareloc,
                                  FLNode->symbol_table);
                fprintf(stdout,"\n");
            }
            if (switches.showSymTab) {
                fprintf(stdout,"%s Symbols:\n",FLNode->inFileName);
                DisplaySymbolTable(stdout,FLNode->symbol_table);
                fprintf(stdout,"\n");
            }
```

```
        }
    }

    if (!files_read) {
        fprintf(stdout,"No files to link. exiting...\n");
        CleanupLnk();
        exit(0);
    }

    mmachine_OPCODE_init();

    /* show entire program disassembly */
    for (FLNode = switches.InputFiles.head;
         FLNode != NULL; FLNode = FLNode->next) {
        if (switches.showInstructions) {
            fprintf(stdout,"Text Segment for %s: \n", FLNode->inFileName);
            PrintTextSegment(FLNode,switches.skip_nops);
        }
        if (switches.show_data_unsorted) {
            fprintf(stdout,"\nData Segment (unsorted) for %s:\n",
                    FLNode->inFileName);
            PrintDataSegment(FLNode);
        }
        if (switches.show_data_sorted) {
            fprintf(stdout,"\nData Segment (sorted) for %s:\n",
                    FLNode->inFileName);
            PrintSortedDataSegment(FLNode);
        }
    }

    switches.InputFiles.segment_names =
      GetSegmentNames(&(switches.InputFiles));

    {
        int x = 0;
        ULong64 u64;
        for (x = 0; x < switches.InputFiles.num_segments; x++)
          fprintf(stdout,"segment name : %s\n",
                  switches.InputFiles.segment_names[x]);
    }

    if (!switches.outputFileName) {
        CleanupLnk();
        exit(0);
    }

    /* now can look at all of the entries...*/
    /* patch SYMBOL TABLE entries for an updated address space */
#if 0
    textoffset = dataoffset = 0;
    for (FLNode = switches.InputFiles.head;
         FLNode != NULL;
         FLNode = FLNode->next) {
        if (!FLNode->inFile) continue;
        /* read each symbol and update TEXT and DATA symbols
           with += size(last segment). */
        update_offsets(FLNode->symbol_table, textoffset, dataoffset);
        textoffset += TrueSize(FLNode);
        dataoffset += FLNode->header.a_data;
    }
#endif
    UpdateSymbolOffsets(switches.InputFiles);

    duplist.num_entries = 0;
    duplist.head = NULL;
    undefList.num_entries = 0;
    undefList.head = NULL;

    if (CheckSymbolTables(switches.InputFiles, &duplist, &undefList)) {
        if (duplist.num_entries) {
            fprintf(stderr,"duplicate symbols defined:\n");
            DisplaySymbolTable(stderr,duplist);
        }

        if (undefList.num_entries) {
            fprintf(stderr,"undefined symbols\n");
            DisplaySymbolTable(stderr,undefList);
        }

        CleanupLnk();
        exit(1);
    }

    /* now that the symbol tables check out, create the combined-symbols
       list */
    CreateCombinedSymbolTable(&(switches.InputFiles));
    if (switches.verbose)
      PrintCombinedSymbolTable(switches.InputFiles.combined_syms,
                               switches.InputFiles.combined_sym_count);

    for (FLNode = switches.InputFiles.head;
         FLNode != NULL;
         FLNode = FLNode->next) {
        if (!FLNode->inFile) continue;
        /* search for undefined values and patch the local symbol tables
           of all of the files with them */
        if (patch_symbols(FLNode->symbol_table, switches.InputFiles,
                          FLNode) == -1)  {
            /* error patching... */
            fprintf(stderr,"lnk: fatal symbol-table patching error.\n"
                    "Please report this bug and save the object files and "
                    "command line options used.\n");
            CleanupLnk();
            exit(1);
        }
        if (switches.showSymTab)
          DisplaySymbolTable(stdout,FLNode->symbol_table);
    }

    /* now all of the values are patched in, so look at relocation
       entries and try to patch all code. */
    if (switches.outputFileName) {
        unlink(switches.outputFileName);
        if ((switches.outputFile = fopen(switches.outputFileName,"w+b"))
            == NULL) {
            fprintf(stderr,"lnk: Could not open %s for output.\n",
                    switches.outputFileName);
            CleanupLnk();
            exit(1);
        }
    } else {
        /* no output was requested */
        CleanupLnk();
        exit(0);
    }

    /* this only works if no symbol problems are evident */
    fprintf(stdout,"Outputting to %s...\n", switches.outputFileName);

    outHeader.a_dynamic = 0;
```

```c
    outHeader.a_toolversion = (switches.debug ?
                               DEBUG_TOOLVERSION : TOOLVERSION);
    outHeader.a_machtype = 1;
    outHeader.a_magic = (unsigned short)(0407); /* SUN Magic Number! */
    outHeader.a_text = outHeader.a_data = outHeader.a_bss = outHeader.a_syms =
      outHeader.a_entry = outHeader.a_trsize = outHeader.a_drsize = 0;
    fwrite(&outHeader,sizeof(outHeader),1,switches.outputFile);

    dataoffset = 0;
    for (FLNode = switches.InputFiles.head;
         FLNode != NULL;
         FLNode = FLNode->next) {
        if (!FLNode->inFile) continue;
        if (!patchHoles(FLNode, outHeader.a_text,
                        dataoffset, switches.verbose)) {
            fprintf(stderr,"lnk: fatal error in symbol patching\n"
                    "lnk aborting.\n");
            CleanupLnk();
            exit(1);
        }
        outHeader.a_text += TrueSize(FLNode);
        dataoffset += FLNode->header.a_data;
        OutputTextSegment(FLNode, switches.outputFile, switches.debug);
        free(FLNode->TextSegment);
    }

    if (switches.debug)
      /* update the a_text to the true size of the text segment in the file */
      outHeader.a_text = (outHeader.a_text * 9L) / 8L;

    /* concatenate the data segments */

    outHeader.a_data = 0;

    {
        int x = 0;
        int y = 0;
        ULong64 u64 = 0;
        for (x = 0; x < switches.InputFiles.num_segments; x++) {
            fprintf(stdout, "%s: %x\n",
                    switches.InputFiles.segment_names[x],
                    switches.InputFiles.segment_offsets[x]);
        }

        if (switches.InputFiles.num_segments) {
            u64 = switches.InputFiles.num_segments * sizeof(u64);
            fwrite(&u64, sizeof(ULong64), 1, switches.outputFile);
            if (switches.InputFiles.num_segments > 1)
              for (y = 1; y < switches.InputFiles.num_segments; y++) {
                  u64 =
                    switches.InputFiles.segment_offsets[y - 1] +
                      switches.InputFiles.num_segments * sizeof(u64);
                  fwrite(&u64, sizeof(ULong64), 1, switches.outputFile);
              }
        }
        outHeader.a_data += switches.InputFiles.num_segments * sizeof(u64);
        if (switches.InputFiles.num_segments)
          outHeader.a_toolversion |= MULTIDATASEG_FLAG;
    }

    for (FLNode = switches.InputFiles.head;
         FLNode != NULL;
         FLNode = FLNode->next) {
        if (!FLNode->inFile) continue;
        outHeader.a_data += FLNode->header.a_data;
        fwrite(FLNode->DataSegment, sizeof(char), FLNode->header.a_data,
               switches.outputFile);
        free(FLNode->DataSegment);
    }

    /* not must output TEXT and DATA relocation to the file */
    outHeader.a_trsize = OutputTextRelocation(switches.InputFiles,
                                              switches.outputFile,
                                              switches.verbose);

    outHeader.a_drsize = OutputDataRelocation(switches.InputFiles,
                                              switches.outputFile);

    outHeader.a_syms = OutputSymbolTable(switches.InputFiles,
                                         switches.outputFile,
                                         &outHeader.a_entry) * sizeof(NList);

    if (switches.verbose)
      fprintf(stdout,"Program entry at address $%08x\n", outHeader.a_entry);
    fseek(switches.outputFile,0,0);
    fwrite(&outHeader,sizeof(outHeader),1,switches.outputFile);
    chmod (switches.outputFileName, 0777 & ~umask (0));
    CleanupLnk();
    return(0);
}

void impl_error() { }
```

```
/* M-Machine linker
   Yevgeny Gurevich, MIT AI Lab - Concurrent VLSI Architecture Group
   Spring, 1994 */

/* linker definitions - used for all derived sources */
#define LNKDEFS_INCLUDED

#ifndef MM_OBJECT_INCLUDED
#include "mm_object.h"
#define MM_OBJECT_INCLUDED
#endif

#define N_SEGMENT 0xc0

#define TrueSize(node) (node->isDebug ? LogicalSize(node->header.a_text) : \
                        node->header.a_text)

#ifndef LINKED_TOOLVERSION
#define LINKED_TOOLVERSION 0x21
#endif

#ifndef MULTIDATASEG_FLAG
#define MULTIDATASEG_FLAG 0x20
#endif

#define isMultiDataSeg(node) (node->header.a_toolversion & MULTIDATASEG_FLAG)

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

#ifndef LONG64
#define LONG64
#endif

#ifndef FREE
#define FREE(x)     if(x) free(x), (x) = NULL
#endif
```

```c
/* M-Machine linker
   Yevgeny Gurevich, MIT AI Lab - Concurrent VLSI Architecture Group
   Spring, 1994 */

/* object_read source - reading object files into data structures */

#ifndef OBJECT_READ_INCLUDED
#include "object_read.h"
#endif

#ifdef __MSDOS__
#include <stdlib.h>
#else
#include <sys/stat.h>
int unlink(char *);
int fclose(FILE *);
int fflush(FILE *);
int fprintf(FILE *, char *, ...);
int fread(void *, size_t, size_t, FILE*);
int fseek(FILE*, long int, int);
int fwrite(void *, int, int, FILE*);
#if 0
void free();
char *malloc(), *calloc();
int exit();
#endif
#endif

/* returns 0 on failure, 1 on success */
int ReadRelocationInformation(FListNode* N) {
    /* reads text and data relocation information from the FListNode file */
    int i;
    Relocation_Entry *e;
    RelocNode *rnode = NULL;
    unsigned long num_entries;

    if (N->inFile == NULL)
      return 0;
    fseek(N->inFile,N_TRELOFF(N->header),0);

    num_entries = N->header.a_trsize / sizeof(Relocation_Entry);
    N->relocation.head = NULL;

    for (i = 0; i < num_entries; i++) {
        if ((e = (Relocation_Entry*)malloc(sizeof(Relocation_Entry)))
            == NULL) {
            fprintf(stderr,"Error allocating relocation information entry.\n");
            return 0;
        }

        if (fread(e, sizeof(Relocation_Entry), 1, N->inFile) != 1) {
            fprintf(stderr,
                    "Error reading relocation information entry %ld.\n", i);
            return 0;
        }

        if ((rnode = RelocEntry2RelocNode(e)) != NULL)
          RelocList_add(&(N->relocation), rnode);
        else {
            fprintf(stderr,
                    "Error adding text reloc information entry %ld.\n", i);
            return 0;
        }
    }

    if (N->relocation.num_entries != num_entries) {
        fprintf(stderr, "Expected and actual text relocation entry counts "
                "do not match\n");
        return 0;
    }

    fseek(N->inFile,N_DRELOFF(N->header),0);

    num_entries = N->header.a_drsize / sizeof(Relocation_Entry);
    N->datareloc.head = NULL;

    for (i = 0; i < num_entries; i++) {
        if ((e = (Relocation_Entry*)malloc(sizeof(Relocation_Entry)))
            == NULL) {
            fprintf(stderr,"Error allocating relocation information entry.\n");
            return 0;
        }
        if (fread(e, sizeof(Relocation_Entry), 1, N->inFile) != 1) {
            fprintf(stderr,
                    "Error reading relocation information entry %ld.\n", i);
            return 0;
        }

        if ((rnode = RelocEntry2RelocNode(e)) != NULL)
          RelocList_add(&(N->datareloc), rnode);
        else {
            fprintf(stderr,
                    "Error adding data reloc information entry %ld.\n", i);
            return 0;
        }
    }

    if (N->datareloc.num_entries != num_entries) {
        fprintf(stderr, "Expected and actual data relocation entry counts "
                "do not match\n");
        return 0;
    }

    return 1;
}

/* returns 0 on failure, 1 on success */
int ReadSymbolTable(FListNode *N) {
    /* reads the symbol table from the FListNode file */
    int i, j;
    SymNode *snode;
    NList *nl;
    char temp[100];
    unsigned long num_entries;

    /* try to read the symbol table stuff! */
    num_entries = N->header.a_syms / sizeof(NList);
    N->symbol_table.head = NULL;

    fseek(N->inFile,N_SYMOFF(N->header),0);

    for (i = 0; i < num_entries; i++) {
        if ((nl = (NList*)malloc(sizeof(NList))) != NULL) {
            if (fread(nl,sizeof(NList),1,N->inFile) != 1) {
                fprintf(stderr,"Error reading symbol table\n");
                return 0;
            }

            if ((snode = SymNode_create()) != NULL)
              snode->entry = nl;
```

```
            SymList_add(&(N->symbol_table), snode);
        }
    }

    if (N->symbol_table.num_entries != num_entries) {
        fprintf(stderr, "Expected and actual symbol_table counts "
                "do not match\n");
        return 0;
    }

    for (snode = N->symbol_table.head, j = 0;
         snode != NULL;
         snode = snode->next, j = 0) {
        fseek(N->inFile,
              N_STROFF(N->header) + snode->entry->n_un.n_strx, 0);

        if (fread(&temp[j++],sizeof(char),1,N->inFile) != 1) {
            fprintf(stderr,"Error reading symbol table\n");
            return 0;
        }

        while(temp[j-1] != '\0') {
            if (fread(&temp[j++],sizeof(char),1,N->inFile) != 1) {
                fprintf(stderr,"Error reading symbol table\n");
                return 0;
            }
        }
        snode->entry->n_un.n_name = (char*)calloc(strlen(temp)+2);
        sprintf(snode->entry->n_un.n_name, temp);
    }

    /* try to set up segment names here */
    for (snode = N->symbol_table.head, N->num_segments = 0 ;
         snode != NULL;
         snode = snode->next) {
        if (snode->entry->n_type == N_SEGMENT) {
            N->num_segments++;
        }
    }

    N->segment_names = (char**)calloc(sizeof(char*),
                                      N->num_segments);
    N->segment_offsets = (unsigned long*)calloc(sizeof(unsigned long*),
                                                N->num_segments);
    for (snode = N->symbol_table.head, j = 0;
         snode != NULL;
         snode = snode->next) {
        if (snode->entry->n_type == N_SEGMENT) {
            N->segment_names[j] =
              (char*)calloc(sizeof(char),
                            snode->entry->n_un.n_name + 1);
            sprintf(N->segment_names[j], snode->entry->n_un.n_name);
            N->segment_offsets[j] = 0;
            j++;
        }
    }
    return 1;
}

/* returns 0 on failure, 1 on success */
int ReadTextSegment(FListNode *FLNode) {
    /* reads the text segment from the FListNode file */
    unsigned long addr;

    if (!FLNode->inFile) return 0;

    if ((FLNode->TextSegment =
         (codebits*)calloc(TrueSize(FLNode) / 4L, sizeof(codebits))) == NULL) {
        fprintf(stderr,"Error allocating text segment.\n");
        return 0;
    }

    FLNode->TextInfo = NULL;

    if (FLNode->isDebug) {
        /* must allocate same space, but read in strange chunks...*/
        unsigned long debug_text_size = FLNode->header.a_text;
        char *tempSegment = NULL;
        if ((tempSegment = (char*)calloc(debug_text_size, sizeof(char),1))
            == NULL) {
            fprintf(stderr,"Error allocating temp text segment.\n");
            return 0;
        }

        FLNode->TextInfo = (unsigned char*)calloc(TrueSize(FLNode) / 4L,
                                                  sizeof(char));

        fseek(FLNode->inFile, N_TXTOFF(FLNode->header), 0);
        if (fread(tempSegment, sizeof(char), debug_text_size, FLNode->inFile)
            != debug_text_size) {
            fprintf(stderr,"Error reading text segment.\n");
            return 0;
        }

        /* now must fill up the TextSegment */
        addr = 0; debug_text_size = 0;
        while (addr < LogicalSize(FLNode->header.a_text) / 4) {
            if (!(addr % 2)) {
                if ((unsigned char)(tempSegment[debug_text_size]) == 0xee) {
                    /* CDATA */
                    FLNode->TextInfo[addr] = 0xEE;

                    memcpy(FLNode->TextSegment + addr ,
                           tempSegment + debug_text_size + 1,
                           sizeof(ULong64));

                    debug_text_size += 9;
                    addr++;
                    addr++;
                    continue;
                } else {
                    FLNode->TextInfo[addr] = tempSegment[debug_text_size];
                }
                debug_text_size++;
            }
            memcpy(FLNode->TextSegment + addr,
                   tempSegment + debug_text_size,
                   sizeof(codebits)/sizeof(char));
            addr++; debug_text_size += 4;
        }
        free(tempSegment);
    } else {
        fseek(FLNode->inFile, N_TXTOFF(FLNode->header), 0);
        if ((fread(FLNode->TextSegment,
                   sizeof(codebits),
                   FLNode->header.a_text / 4L,
                   FLNode->inFile)) != (FLNode->header.a_text / 4L)) {
            fprintf(stderr,"Error reading text segment\n");
            return 0;
        }
    }
```

```
    return 1;
}

/* returns 0 on failure, 1 on success */
int ReadDataSegment(FListNode *FLNode) {
    /* reads the data segment from the FListNode file */
    ULong64 u64;
    int x;

    if (!FLNode->inFile) return 0;
    if ((FLNode->DataSegment =
         (char *)calloc(FLNode->header.a_data,
                        sizeof(char))) == NULL) {
        fprintf(stderr,"Error allocating data segment.\n");
        return 0;
    }
    fseek(FLNode->inFile,N_DATOFF(FLNode->header),0);

    if (isMultiDataSeg(FLNode)) {
        for (x = 0; x < FLNode->num_segments; x++) {
            if (fread(&u64,sizeof(u64),1,FLNode->inFile) != 1)
              return 0;
            FLNode->segment_offsets[x] = (unsigned long)u64;
        }

        fseek(FLNode->inFile,N_DATOFF(FLNode->header),0);
    }

    if (fread(FLNode->DataSegment, sizeof(char), FLNode->header.a_data,
              FLNode->inFile) != FLNode->header.a_data) {
        fprintf(stderr,"Error reading data segment.\n");
        return 0;
    }
    return 1;
}

/*************************************
  High-Level Read Function
  ************************************/

int ReadObjectFile(FListNode *FLNode) {
    /* read the header */
    if ((!FLNode) || (!(FLNode->inFile)))
      return 0;

    fread(&(FLNode->header), sizeof(struct exec), 1, FLNode->inFile);
    if (IsDebugObjectFile(FLNode->header.a_toolversion))
      FLNode->isDebug = TRUE;
    else FLNode->isDebug = FALSE;
    if (FLNode->header.a_toolversion == LINKED_TOOLVERSION)
      FLNode->isLinked = TRUE;
    else FLNode->isLinked = FALSE;

    FLNode->relocation.num_entries = 0;
    FLNode->datareloc.num_entries = 0;
    FLNode->symbol_table.num_entries = 0;

    return (ReadRelocationInformation(FLNode) &&
            ReadSymbolTable(FLNode) &&
            ReadTextSegment(FLNode) &&
            ReadDataSegment(FLNode));
}
```

```
/* M-Machine linker
   Yevgeny Gurevich, MIT AI Lab - Concurrent VLSI Architecture Group
   Spring, 1994 */

/* object_read header - reading object files into data structures */
#define OBJECT_READ_INCLUDED

#ifndef DATASTRUCT_INCLUDED
#include "datastruct.h"
#endif

#ifndef LNKDEFS_INCLUDED
#include "lnkdefs.h"
#endif

int ReadObjectFile(FListNode*);
int ReadRelocationInformation(FListNode*);
int ReadSymbolTable(FListNode*);
int ReadTextSegment(FListNode*);
int ReadDataSegment(FListNode*);
```

```c
/* object_unparse.c -- display functions for object-file data structures. */

#ifndef OBJECT_UNPARSE_INCLUDED
#include "object_unparse.h"
#endif

#ifdef __MSDOS__
#include <stdlib.h>
#else
#include <sys/stat.h>
int unlink(char *);
int fclose(FILE *);
int fflush(FILE *);
int fprintf(FILE *, char *, ...);
int fseek(FILE*, long int, int);
int fwrite(void *, int, int, FILE*);
#if 0
void free();
char *malloc(), *calloc();
int exit();
#endif
#endif

int isDebugDataReloc(int type) {
    switch ((MM_RELOCATION)type) {
    case DATA_ASCIIZ_RELOC:
    case DATA_CHAR_RELOC:
    case DATA_I8_RELOC:
    case DATA_I16_RELOC:
    case DATA_I32_RELOC:
    case DATA_I64_RELOC:
    case DATA_F32_RELOC:
    case DATA_F64_RELOC:
    case DATA_PTR_RELOC:
    case DATA_OTHER_RELOC:
        return 1;
    default:
        return 0;
    }
}

char *Ntype2String(long int t) {
    static char foo[100] = "";

    foo[0] = '\0';

    if (t & N_EXT) {
        t -= N_EXT;
        if (t == N_DATA)
          strcat(foo,"D");
        else if (t == N_ABS)
          strcat(foo,"A");
        else if (t == N_TEXT)
          strcat(foo,"T");
        else if (t == N_UNDF)
          strcat(foo,"U");
    } else {
        if (t == N_DATA)
          strcat(foo,"d");
        else if (t == N_ABS)
          strcat(foo,"a");
        else if (t == N_TEXT)
          strcat(foo,"t");
        else if (t == N_UNDF)
          strcat(foo,"u");
        else if (t == 0xc0)
          strcat(foo,"S");
    }
    return foo;
}

char *RelocType2String(int type) {
    switch ((MM_RELOCATION)type) {
    case ABS_AECI_RELOC:     return "AECI Relocation  7-bit";
    case ABS_B_RELOC:        return "B    Relocation 11-bit";
    case ABS_G_RELOC:        return "Absolute G Rel. 18-bit";
    case REL_G_RELOC:        return "Relative G Rel. 18-bit";
    case DATA_ASCIIZ_RELOC:  return "ASCII terminated string";
    case DATA_CHAR_RELOC:    return "ASCII character";
    case DATA_I8_RELOC:      return "8-bit Int";
    case DATA_I16_RELOC:     return "16-bit Int";
    case DATA_I32_RELOC:     return "32-bit Int";
    case DATA_I64_RELOC:     return "64-bit Int";
    case DATA_F32_RELOC:     return "32-bit Float";
    case DATA_F64_RELOC:     return "64-bit Float";
    case DATA_PTR_RELOC:     return "64-bit Pointer";
    case DATA_OTHER_RELOC:   return "Other";
    default:                 return "unknown";
    }
}

void PrintStabEntry(FILE *out, NList *nl) {
    fprintf(out,"%20s\t", nl->n_un.n_name);
    fprintf(out,"%s\t", Ntype2String(nl->n_type));
    fprintf(out,"0x%08x\n", nl->n_value);
}

void DisplaySymbolTable(FILE *out, SymList L) {
    SymNode *snode;

    for (snode = L.head; snode != NULL; snode = snode->next) {
        PrintStabEntry(out,snode->entry);
    }
}

void DisplayUndefinedSymbols(FILE *out, inFileList flist) {
    FListNode *FLNode;
    SymNode *snode;
    for (FLNode = flist.head; FLNode != NULL; FLNode = FLNode->next) {
        for (snode = FLNode->symbol_table.head;
             snode != NULL;
             snode = snode->next)
          if (isUndefSymbol(*snode->entry))
            PrintStabEntry(out,snode->entry);
    }
}

void DisplayRelocation(FILE *out, RelocList R, SymList S) {
    RelocNode *rnode;
    SymNode *snode;
    long int sgn = 0;

    fprintf(out,"\t# Reloc entries = %ld\n",R.num_entries);

    for (rnode = R.head; rnode != NULL; rnode = rnode->next) {
        sgn = rnode->entry->r_index; if (sgn == 0xffffff) sgn = -1;

        fprintf(out,"\t\t$%08x: %25s\t%ld",
                rnode->entry->r_address,
                RelocType2String(rnode->entry->r_type),
```

```
                sgn);

        if (!isDebugDataReloc(rnode->entry->r_type)) {
            if ((snode = FindSymTabEntry(S, rnode->entry->r_index)) != NULL)
              fprintf(out,"\t%s\n", snode->entry->n_un.n_name);
            else
              fprintf(out,"???\n");
        } else if (sgn == -1L) {
            fprintf(out,"\t<ABS>\n");
        } else {
            if ((snode = FindSymTabEntry(S, rnode->entry->r_index)) != NULL)
              fprintf(out,"\t%s\n", snode->entry->n_un.n_name);
            else
              fprintf(out,"???\n");
        }
    }
}

void DisplayHeader(FILE *out, struct exec H) {
    switch (H.a_toolversion & 0x4f) {
    case DEBUG_TOOLVERSION:
        fprintf(out,"DEBUG-format object file\n");
        break;
    case TOOLVERSION:
        break;
        /* case LINKED_TOOLVERSION:
           fprintf(out,"LINKED-format object file\n");
           break; */
    default:
        fprintf(out,"Unknown version object file.\n");
        break;
    }

    if (H.a_toolversion | MULTIDATASEG_FLAG)
      fprintf(out,"Multi-Dataseg file.\n");

    fprintf(out,"Text Segment Size = %d (%d instructions)\n",
            H.a_text, H.a_text / 4);
    if (IsDebugObjectFile(H.a_toolversion))
      fprintf(out,"\t%d real ops\n",LogicalSize(H.a_text));
    fprintf(out,"Data Segment Size = %d\n",H.a_data);
    fprintf(out,"Number of Symbols = %d\n",H.a_syms / sizeof(NList));
    fprintf(out,"Number of Text Relocs  = %d\n",
            H.a_trsize / sizeof(Relocation_Entry));
    fprintf(out,"Number of Data Relocs  = %d\n",
            H.a_drsize / sizeof(Relocation_Entry));
}

void print_pretty_ptr(FILE *out, Mas_Pointer ptr) {
    switch(ptr.prot) {
    case P_READ:      fprintf(out,"r."); break;
    case P_RW:        fprintf(out,"rw."); break;
    case P_EXUSER:    fprintf(out,"xu."); break;
    case P_EXSYS:     fprintf(out,"xp."); break;
    case P_ENTERUSER: fprintf(out,"eu."); break;
    case P_ENTERSYS:  fprintf(out,"ep."); break;
    case P_EXMSG:     fprintf(out,"xm."); break;
    case P_CONFIG:    fprintf(out,"c."); break;
    case P_KEY:       fprintf(out,"k."); break;
    case P_PHYSICAL:  fprintf(out,"p."); break;
    case P_ERROR:
    default:
        fprintf(out,"er."); break;
        break;
    }
    fprintf(out,"%02x.%s",ptr.length,CvtXL64((ULong64)ptr.address));
}

void PrintDebuggedDataSegment(FListNode *FLNode) {
    RelocList R = FLNode->datareloc;
    char *DataSegment = FLNode->DataSegment;
    RelocNode *rnode;
    char *datasegPtr;
    unsigned short u16;
    unsigned long u32;
    ULong64 u64;
    float f32;
    double f64;
    Mas_Pointer ptr;

    if (isMultiDataSeg(FLNode)) {
        /* display data segment offset values */
        fprintf(stdout,"Multiple Data Segment Starting Points: \n");
        for (u32 = 0; u32 < FLNode->num_segments; u32++)
          fprintf(stdout,"%s -- %x\n",
                  FLNode->segment_names[u32],
                  FLNode->segment_offsets[u32]);
        fprintf(stdout,"\n");
    }
#if 0
        for (FLNode = switches.InputFiles.head;
             FLNode != NULL;
             FLNode = FLNode->next) {
            if (isMultiDataSeg(FLNode)) {
                fseek(FLNode->inFile,N_DATOFF(FLNode->header),0);
                for (x = 0; x < switches.InputFiles.num_segments; x++) {
                    fread(&u64,sizeof(u64),1,FLNode->inFile);
                    fprintf(stdout,"%s -- %s\n",
                            switches.InputFiles.segment_names[x],
                            CvtXL64(u64));
                }
            }
        }
#endif

    for (rnode = R.head; rnode != NULL; rnode = rnode->next) {
        datasegPtr = DataSegment + rnode->entry->r_address;
        /* if (isMultiDataSeg(FLNode))
          datasegPtr += FLNode->segment_offsets[0]; */

        fprintf(stdout,"$%08x: ", rnode->entry->r_address);
        switch(rnode->entry->r_type) {
        case DATA_CHAR_RELOC:
            fprintf(stdout,"<CHAR>  \t\"%c\"\n",(char)datasegPtr[0]);
            break;
        case DATA_ASCIIZ_RELOC:
            fprintf(stdout,"<STRING>\t\"%s\"\n",datasegPtr);
            break;
        case DATA_I8_RELOC:
            fprintf(stdout,"<I8>   \t0x%02x      \t%d\n",
                    (unsigned char)datasegPtr[0],
                    (unsigned char)datasegPtr[0]);
            break;
        case DATA_I16_RELOC:
            memcpy(&u16,datasegPtr,sizeof(u16));
            fprintf(stdout,"<I16>  \t0x%04x    \t%d\n", u16, u16);
            break;
        case DATA_I32_RELOC:
            memcpy(&u32,datasegPtr,sizeof(u32));
            fprintf(stdout,"<I32>  \t0x%08x\t%ld\n",u32, u32);
```

```
            break;
        case DATA_I64_RELOC:
            memcpy(&u64,datasegPtr,sizeof(u64));
            fprintf(stdout,"<I64>  \t0x%s\n",CvtXL64(u64));
            break;
        case DATA_F32_RELOC:
            memcpy(&f32,datasegPtr,sizeof(f32));
            fprintf(stdout,"<F32>  \t%f\n",f32);
            break;
          case DATA_F64_RELOC:
            memcpy(&f64,datasegPtr,sizeof(f64));
            fprintf(stdout,"<F64>  \t%f\n",f64);
            break;
        case DATA_PTR_RELOC:
            memcpy(&ptr,datasegPtr,sizeof(ptr));
            fprintf(stdout,"<PTR>\t");
            print_pretty_ptr(stdout,ptr);
            fprintf(stdout,"\n");
            break;
        default:
            break;
        }
    }
}

void PrintDataSegment(FListNode *FLNode) {
    unsigned long addr = 0, dat = 0;

    if (FLNode->isDebug || FLNode->isLinked) {
        /* can do this more intellignly */
        PrintDebuggedDataSegment(FLNode);
    } else {
        while (addr < FLNode->header.a_data) {
            memcpy(&dat,FLNode->DataSegment+addr,4);
            fprintf(stdout,"$%08x: %08x\n", addr, dat);
            addr += 4;
        }
    }
}

void PrintSortedDataSegment(FListNode *FLNode) {
    /* How about deallocation ?? */
    SegmentInfo sinfo;
    DataEntry *entry;

    if (FLNode->isDebug || FLNode->isLinked) {
        /* can do this more intelligently */
        fprintf(stdout,"--- Data Segment:\nInts:\n");
        for (sinfo = GetData(TAG_INT, FLNode), entry = sinfo.head;
             entry != NULL; entry = entry->next)
          fprintf(stdout,"\t$%08x: %s\n", entry->address,
                  CvtXL64(entry->info.u64));
        fprintf(stdout,"Floats:\n");
        for (sinfo = GetData(TAG_FLOAT, FLNode), entry = sinfo.head;
             entry != NULL; entry = entry->next)
          fprintf(stdout,"\t$%08x: %f\n", entry->address, entry->info.f64);
        fprintf(stdout,"Ptrs:\n");
        for (sinfo = GetData(TAG_MAP_PTR, FLNode), entry = sinfo.head;
             entry != NULL; entry = entry->next) {
            fprintf(stdout,"\t$%08x: ", entry->address);
            print_pretty_ptr(stdout,entry->info.ptr);
            fprintf(stdout,"\n");
        }
        fprintf(stdout,"Strings:\n");
        for (sinfo = GetData(TAG_STRING, FLNode), entry = sinfo.head;
             entry != NULL; entry = entry->next) {
            fprintf(stdout,"\t$%08x: %s\n", entry->address,
                    entry->info.string);
        }

        fprintf(stdout,"\n--- Code Data Segment:\nInts:\n");
        for (sinfo = GetCData(TAG_INT, FLNode), entry = sinfo.head;
             entry != NULL; entry = entry->next)
          fprintf(stdout,"\t$%08x: %s\n", entry->address,
                  CvtXL64(entry->info.u64));
        fprintf(stdout,"Floats:\n");
        for (sinfo = GetCData(TAG_FLOAT, FLNode), entry = sinfo.head;
             entry != NULL; entry = entry->next)
          fprintf(stdout,"\t$%08x: %f\n", entry->address, entry->info.f64);
        fprintf(stdout,"Ptrs:\n");
        for (sinfo = GetCData(TAG_MAP_PTR, FLNode), entry = sinfo.head;
             entry != NULL; entry = entry->next) {
            fprintf(stdout,"\t$%08x: ", entry->address);
            print_pretty_ptr(stdout,entry->info.ptr);
            fprintf(stdout,"\n");
        }
        fprintf(stdout,"Strings:\n");
        for (sinfo = GetCData(TAG_STRING, FLNode), entry = sinfo.head;
             entry != NULL; entry = entry->next) {
            fprintf(stdout,"\t$%08x: %s\n", entry->address,
                    entry->info.string);
        }
    } else
      fprintf(stderr,
              "Cannot show sorted data segment for non-dubugged file.\n");
}

RelocNode *FindRelocationEntry(RelocList L, unsigned long addr) {
    RelocNode *rnode;

    for (rnode = L.head; rnode != NULL; rnode = rnode->next) {
        if (rnode->entry->r_address == addr)
          return rnode;
    }
    return NULL;
}

void PrintTextSegment(FListNode *FLNode, char skip_nops) {
    MM_FUNCUNIT current_funit = IALU;
    MM_FUNCUNIT next_funit = IALU;
    char new_instr = TRUE;
    char imf_seen  = FALSE;
    unsigned long addr, next_addr;
    codebits *CPtr;
    codebits *CNext;
    OpBufEl O;
    RelocNode *rnode = NULL;

    ULong64     u64;
    double      f64;
    Mas_Pointer ptr;


    for (addr = 0L; addr < TrueSize(FLNode); addr += 4L) {
        if (!(addr % 8)) {
            if (FLNode->TextInfo && (FLNode->TextInfo[addr / 4L] == 0xEE)) {
                /* code data */
                /* find out which type of data it is...*/
                rnode = FindRelocationEntry(FLNode->relocation, addr);
```

```
            if (!rnode) {
                memcpy(&u64,FLNode->TextSegment + (addr / 4L),sizeof(u64));
                fprintf(stdout,"$%08x: [DAT ERR ] 0x%s\n",
                        addr, CvtXL64(u64));
            } else {
                switch(rnode->entry->r_type) {
                case DATA_I64_RELOC:
                    memcpy(&u64,FLNode->TextSegment + (addr / 4L),
                           sizeof(u64));
                    fprintf(stdout,"$%08x: [INT     ] 0x%s\n",
                            addr, CvtXL64(u64));
                    break;
                case DATA_F64_RELOC:
                    memcpy(&f64,FLNode->TextSegment + (addr / 4L),
                           sizeof(f64));
                    fprintf(stdout,"$%08x: [FP      ] %f\n", addr, f64);
                    break;
                case DATA_PTR_RELOC:
                    memcpy(&ptr,FLNode->TextSegment + (addr / 4L),
                           sizeof(ptr));
                    fprintf(stdout,"$%08x: [PTR     ] ", addr);
                    print_pretty_ptr(stdout, ptr);
                    fprintf(stdout,"\n");
                    break;
                default:
                    memcpy(&u64,FLNode->TextSegment + (addr / 4L),
                           sizeof(u64));
                    fprintf(stdout,"$%08x: [DAT ??? ] 0x%s\n",
                            addr, CvtXL64(u64));
                }
            }
            addr += 4L;
            continue;
        }
    }

    if (addr % 8L)
      CPtr = &(FLNode->TextSegment[ (addr - (addr % 8L)) / 4L]);
    else
      CPtr = &(FLNode->TextSegment[ (addr / 4L) + 1L ]);

    next_addr = addr + 4;
    if (next_addr < TrueSize(FLNode))
      if (next_addr % 8L)
        CNext = &(FLNode->TextSegment[(next_addr -
                                       (next_addr % 8L)) / 4L]);
      else
        CNext = &(FLNode->TextSegment[(next_addr / 4L) + 1L ]);
    else
      CNext = NULL;

    if ((KillPackBits(Bits2Long(*CPtr)) != IALU_NOP_MASK) ||
        ((KillPackBits(Bits2Long(*CPtr)) == IALU_NOP_MASK) &&
         !skip_nops)) {
        fprintf(stdout,"$%08x: [%08x] ", addr, Bits2Long(*CPtr));
        OpBufEl_assembler_init(&O);

        /* now, which pack info is this ? */

        if (new_instr) {
            switch (GetPackBits(Bits2Long(*CPtr))) {
            case IALU_ONLY:
                current_funit = IALU;
                new_instr = TRUE;
                break;
            case FALU_ONLY:
                current_funit = FALU;
                new_instr = TRUE;
                break;
            case MEMU_ONLY:
                current_funit = MEMU;
                new_instr = TRUE;
                break;
            case MULT:
                /* must look at NEXT pack bits */
                new_instr = FALSE;
                imf_seen  = FALSE;
                if (!CNext) {
                    fprintf(stderr, "Reached end of instructions "
                            "with ambiguity\n");
                    return;
                }
                switch (GetPackBits(Bits2Long(*CNext))) {
                case IM_COMBO:
                    current_funit = IALU;
                    next_funit = MEMU;
                    break;
                case IF_COMBO:
                    current_funit = IALU;
                    next_funit = FALU;
                        break;
                case MF_COMBO:
                    current_funit = MEMU;
                    next_funit = FALU;
                    break;
                case IMF_COMBO:
                    current_funit = IALU;
                    next_funit = MEMU;
                    imf_seen = TRUE;
                    break;
                default:
                    fprintf(stderr,"pack bits 2 impl error\n");
                    return;
                }
                break;
            default:
                fprintf(stderr,"pack bits 1 impl error\n");
                return;
            }
        } else {
            current_funit = next_funit;
            if (imf_seen) {
                imf_seen = FALSE;
                next_funit = FALU;
                } else
                  new_instr = TRUE;
        }

        if (OpBufEl_unpack(&O, current_funit, *CPtr) != UNPACK_ERROR)
          OpBufEl_operation_print(&O, stdout, current_funit);
        else fprintf(stdout,"???\n");
    }
  }
}

void AddDataToSegInfo(SegmentInfo *seg, DataEntry *entry) {
    if (seg->head == NULL)
      seg->head = seg->current = entry;
    else {
        seg->current->next = entry;
```

```
        seg->current = entry;
    }
}

SegmentInfo GetCData(int type, FListNode *FLNode) {
    RelocNode *rnode;
    RelocList RR = FLNode->relocation;
    codebits *textSegPtr;
    DataEntry* entry;
    SegmentInfo seginfo;

    ULong64 u64;
    double  f64;
    Mas_Pointer ptr;

    seginfo.head = NULL;
    for (rnode = RR.head; rnode != NULL; rnode = rnode->next) {
        textSegPtr = FLNode->TextSegment + (rnode->entry->r_address / 4L);
        switch(rnode->entry->r_type) {
        case DATA_I64_RELOC:
            if (type == TAG_INT) {
                entry = (DataEntry*)malloc(sizeof(DataEntry));
                entry->next = NULL;
                memcpy(&u64,textSegPtr,sizeof(u64));
                entry->info.u64 = u64;
                entry->tag = TAG_INT;
                entry->address = rnode->entry->r_address;
                AddDataToSegInfo(&seginfo,entry);
            }
            break;
        case DATA_F64_RELOC:
            if (type == TAG_FLOAT) {
                entry = (DataEntry*)malloc(sizeof(DataEntry));
                entry->next = NULL;
                memcpy(&f64,textSegPtr,sizeof(f64));
                entry->info.f64 = f64;
                entry->tag = TAG_FLOAT;
                entry->address = rnode->entry->r_address;
                AddDataToSegInfo(&seginfo,entry);
            }
            break;
        case DATA_PTR_RELOC:
            if (type == TAG_MAP_PTR) {
                entry = (DataEntry*)malloc(sizeof(DataEntry));
                entry->next = NULL;
                memcpy(&ptr,textSegPtr,sizeof(ptr));
                entry->info.ptr = ptr;
                entry->tag = TAG_MAP_PTR;
                entry->address = rnode->entry->r_address;
                AddDataToSegInfo(&seginfo,entry);
            }
            break;
        case DATA_ASCIIZ_RELOC:
            /* NEED TO BE CHANGED *** ??? */
            if (type == TAG_STRING) {
                entry = (DataEntry*)malloc(sizeof(DataEntry));
                entry->next = NULL;
                entry->info.string = (char*)textSegPtr;
                entry->tag = TAG_STRING;
                entry->address = rnode->entry->r_address;
                AddDataToSegInfo(&seginfo,entry);
            }
            break;
        default:
            break;
            /*
              entry->info.string = dataSegPtr;
              entry->tag = TAG_ERROR;
              entry->address = rnode->entry->r_address;
              AddDataToSegInfo(&seginfo,entry);
              */
            break;
        }
    }
    return seginfo;
}

SegmentInfo GetData(int type, FListNode *FLNode) {
    RelocNode *rnode;
    RelocList R = FLNode->datareloc;
    char *dataSegPtr;
    DataEntry* entry;
    SegmentInfo seginfo;

    ULong64 u64;
    double  f64;
    Mas_Pointer ptr;
    unsigned long u32;
    seginfo.head = NULL;

#if 0
    if ((type == TAG_INT) && isMultiDataSeg(FLNode)) {
        for (u32 = 0; u32 < FLNode->num_segments; u32++) {
            dataSegPtr = FLNode->DataSegment + (u32 * 8);
            entry = (DataEntry*)malloc(sizeof(DataEntry));
            entry->next = NULL;
            memcpy(&u64,dataSegPtr,sizeof(u64));
            entry->info.u64 = u64;
            entry->tag = TAG_INT;
            entry->address = (u32 * 8);
            AddDataToSegInfo(&seginfo,entry);
        }
    }
#endif

    for (rnode = R.head; rnode != NULL; rnode = rnode->next) {
        dataSegPtr = FLNode->DataSegment + rnode->entry->r_address;
        /* if (isMultiDataSeg(FLNode))
           dataSegPtr += FLNode->segment_offsets[0]; */
        switch(rnode->entry->r_type) {
        case DATA_I64_RELOC:
            if (type == TAG_INT) {
                entry = (DataEntry*)malloc(sizeof(DataEntry));
                entry->next = NULL;
                memcpy(&u64,dataSegPtr,sizeof(u64));
                entry->info.u64 = u64;
                entry->tag = TAG_INT;
                entry->address = rnode->entry->r_address;
                AddDataToSegInfo(&seginfo,entry);
            }
            break;
        case DATA_F64_RELOC:
            if (type == TAG_FLOAT) {
                entry = (DataEntry*)malloc(sizeof(DataEntry));
                entry->next = NULL;
                memcpy(&f64,dataSegPtr,sizeof(f64));
                entry->info.f64 = f64;
                entry->tag = TAG_FLOAT;
                entry->address = rnode->entry->r_address;
                AddDataToSegInfo(&seginfo,entry);
```

```
            }
            break;
        case DATA_PTR_RELOC:
            if (type == TAG_MAP_PTR) {
                entry = (DataEntry*)malloc(sizeof(DataEntry));
                entry->next = NULL;
                memcpy(&ptr,dataSegPtr,sizeof(ptr));
                entry->info.ptr = ptr;
                entry->tag = TAG_MAP_PTR;
                entry->address = rnode->entry->r_address;
                AddDataToSegInfo(&seginfo,entry);
            }
            break;
        case DATA_ASCIIZ_RELOC:
            if (type == TAG_STRING) {
                entry = (DataEntry*)malloc(sizeof(DataEntry));
                entry->next = NULL;
                entry->info.string = dataSegPtr;
                entry->tag = TAG_STRING;
                entry->address = rnode->entry->r_address;
                AddDataToSegInfo(&seginfo,entry);
            }
            break;
        default:
            break;
            /*
              entry->info.string = dataSegPtr;
              entry->tag = TAG_ERROR;
              entry->address = rnode->entry->r_address;
              AddDataToSegInfo(&seginfo,entry);
              */
              break;
        }
    }
    return seginfo;
}
```

```
/* object_unparse.h -- header file for object-file information-printing
   functions */
#define OBJECT_UNPARSE_INCLUDED

#ifndef LNKDEFS_INCLUDED
#include "lnkdefs.h"
#endif

#ifndef DATASTRUCT_INCLUDED
#include "datastruct.h"
#endif

#include <string.h>

typedef struct data_entry {
    int tag;
    union {
        ULong64 u64;
        double  f64;
        Mas_Pointer ptr;
        char*   string;
    } info;
    unsigned long address;
    struct data_entry *next;
} DataEntry;

typedef struct {
    DataEntry* head;
    DataEntry* current;
} SegmentInfo;

typedef enum {
    IALU_ONLY,
    MEMU_ONLY,
    FALU_ONLY,
    MULT
  } unpack_bits_one;

typedef enum {
    IM_COMBO,
    IF_COMBO,
    MF_COMBO,
    IMF_COMBO
  } unpack_bits_two;

#define GetPackBits(x) ((x & 0xc0000000) >> 30)

void        PrintDataSegment(FListNode*);
void        PrintSortedDataSegment(FListNode*);
void        PrintTextSegment(FListNode*, char);
void        PrintStabEntry(FILE*, NList*);

void        DisplayRelocation(FILE*, RelocList, SymList);
void        DisplayHeader(FILE*, struct exec);
void        DisplaySymbolTable(FILE*, SymList);
void        DisplayUndefinedSymbols(FILE*, inFileList);


SegmentInfo GetData(int, FListNode*);  /* return a list of data words */
SegmentInfo GetCData(int, FListNode*); /* return a list of data words */
```

```
/* M-Machine linker
   Yevgeny Gurevich, MIT AI Lab - Concurrent VLSI Architecture Group
   Spring, 1994 */

/* object_write souce - writing data structures out to disk */

#ifndef OBJECT_WRITE_INCLUDED
#include "object_write.h"
#endif

#ifdef __MSDOS__
#include <stdlib.h>
#else
#include <sys/stat.h>
int unlink(char *);
int fclose(FILE *);
int fflush(FILE *);
int fprintf(FILE *, char *, ...);
int fread(void *, size_t, size_t, FILE*);
int fseek(FILE*, long int, int);
int fwrite(void *, int, int, FILE*);
#if 0
void free();
char *malloc(), *calloc();
int exit();
#endif
#endif

/********************************\
 * executable-creation functions *
\********************************/

void OutputTextSegment(FListNode *FLNode, FILE *out, char debug) {
    /* write the text segment of FLNode to file out.  If the debug
       flag is set, add debugging information to the text segment */
    unsigned char c = 0xff;
    unsigned long i = 0;
    if (debug) {
        for (i = 0; i < ((TrueSize(FLNode) / 4L) - 1); i+=2) {
            c = FLNode->TextInfo[i];
            fwrite(&c,sizeof(char),1,out);
            fwrite(&FLNode->TextSegment[i],sizeof(codebits),1,out);
            fwrite(&FLNode->TextSegment[i+1],sizeof(codebits),1,out);
        }
    } else {
        fwrite(FLNode->TextSegment, sizeof(codebits),
               TrueSize(FLNode) / 4L, out);
    }
}

unsigned long OutputTextRelocation(inFileList FList, FILE *out,
                                   unsigned char verbosemode) {
    FListNode *FLNode;
    unsigned long last_entry, last_address, tr_size;
    RelocNode *rnode;
    SymNode *snode;

    for (FLNode = FList.head, last_entry = last_address = tr_size = 0;
         FLNode != NULL;
         FLNode = FLNode->next) {
        /* output the relocation, only add symbol offsets properly...*/
        /* this will force duplicate symbols, but these duplicates
           will be local only */
        if (verbosemode)
          fprintf(stdout,"relocation patching for %s\n", FLNode->inFileName);
        for (rnode = FLNode->relocation.head;
             rnode != NULL;
             rnode = rnode->next) {
            if (rnode->entry->r_index == 0xffffff) ;
            else {
                /* find the symbol in the combined_symbol_table */
                if ((snode = FindSymTabEntry(FLNode->symbol_table,
                                             rnode->entry->r_index)) != NULL) {
#ifdef DEBUG
                    fprintf(stdout,"%s (%d) -- ",
                            snode->entry->n_un.n_name,
                            rnode->entry->r_index);
#endif

                    rnode->entry->r_index =
                      FindSymbolInCombined(snode->entry->n_un.n_name,
                                           FList.combined_syms,
                                           FList.combined_sym_count);
#ifdef DEBUG
                    fprintf(stdout,"%s (%d)\n",
                            combined_symbols[rnode->entry->r_index],
                            rnode->entry->r_index);
#endif
                } else {
                    fprintf(stderr,"Symbol not found in CST\n");
                    rnode->entry->r_index = 0xffffff;
                }
            }
            rnode->entry->r_address += last_address;
            fwrite(rnode->entry,sizeof(*(rnode->entry)), 1, out);
            tr_size += sizeof(*(rnode->entry));
        }
        last_entry += FLNode->symbol_table.num_entries;
        last_address += TrueSize(FLNode);
    }
    return tr_size;
}

unsigned long OutputDataRelocation(inFileList FList, FILE *out) {
    FListNode *FLNode;
    unsigned long last_entry, last_address, dr_size;
    RelocNode *rnode;
    SymNode *snode;
    Relocation_Entry e;
    int i;

    last_address = FList.num_segments * 8;
    for (FLNode = FList.head, last_entry = dr_size = 0;
         FLNode != NULL;
         FLNode = FLNode->next) {
        for (rnode = FLNode->datareloc.head;
             rnode != NULL;
             rnode = rnode->next) {

            if (rnode->entry->r_index == 0xffffff) ;
            else {
                /* find the symbol in the combined_symbol_table */
                if ((snode = FindSymTabEntry(FLNode->symbol_table,
                                             rnode->entry->r_index)) != NULL) {
                    rnode->entry->r_index =
                      FindSymbolInCombined(snode->entry->n_un.n_name,
                                           FList.combined_syms,
                                           FList.combined_sym_count);
                } else {
                    fprintf(stderr,"Symbol not found in CST\n");
```

```
                    rnode->entry->r_index = 0xffffff;
                }
            }
            rnode->entry->r_address += last_address;
            fwrite(rnode->entry, sizeof(*(rnode->entry)), 1, out);
            dr_size += sizeof(*(rnode->entry));
        }
        last_entry += FLNode->symbol_table.num_entries;
        last_address += FLNode->header.a_data;
    }

    for (i = 0; i < FList.num_segments; i++) {
        e.r_address = i * 8;
        e.r_index   = 0xffffff;
        e.r_type    = DATA_I64_RELOC;

        fwrite(&e, sizeof(e), 1, out);
        dr_size += sizeof(e);
    }
    return dr_size;
}

unsigned long OutputSymbolTable(inFileList FList, FILE *out,
                                unsigned long *entrypoint) {
    /* output symbol-table entries */
    unsigned long sym_index = 4, a_syms = 0, i;
    int string_length = 0;
    SymNode *snode;
    FListNode *FLNode;
    char dup_symbol;
    char **sym_names = FList.combined_syms;

    /* allocate an array of symbols-name strings */

    /* must try to get rid of duplicate global symbols */
    for (FLNode = FList.head; FLNode != NULL; FLNode = FLNode->next) {
        for (snode = FLNode->symbol_table.head;
             snode != NULL;
             snode = snode->next) {
            string_length = strlen(snode->entry->n_un.n_name) + 1;
            for (i = 0, dup_symbol = -1; i < a_syms; i++)
              if (!strcmp(sym_names[i], snode->entry->n_un.n_name) &&
                  (isExternalSymbol(*(snode->entry)) ||
                   isSegmentSymbol(*(snode->entry)))) {
                  /* found an existing symbol already in the symbol table */
                  dup_symbol = i;
                  break;
              }

            if (dup_symbol == -1) {
                if (strcmp(sym_names[a_syms], snode->entry->n_un.n_name)) {

                    fprintf(stderr,"lnk: implementation error - "
                            "symbol names do not match\n");
                    return 0;
                }

                if (!strcmp(snode->entry->n_un.n_name,"Main")) {
                    /* entry point...*/
                    *entrypoint = snode->entry->n_value;
                }

                snode->entry->n_un.n_strx = sym_index;
                fwrite(snode->entry, sizeof(*(snode->entry)), 1, out);
                sym_index += string_length;
                a_syms++;
            }
        }
    }

    /* now must output the strings */
    fwrite(&sym_index, sizeof(sym_index), 1, out);
    for (i = 0; i < a_syms; i++) {
        fwrite(sym_names[i], strlen(sym_names[i]) + 1, 1, out);
    }
    return a_syms;
}
```

```
/* M-Machine linker
   Yevgeny Gurevich, MIT AI Lab - Concurrent VLSI Architecture Group
   Spring, 1994 */

/* object_write header - writing data structures out to disk. */
#define OBJECT_WRITE_INCLUDED

#include <stdio.h>

#ifndef DATASTRUCT_INCLUDED
#include "datastruct.h"
#endif

#ifndef LNKDEFS_INCLUDED
#include "lnkdefs.h"
#endif

void          OutputTextSegment(FListNode*, FILE*, char);
unsigned long OutputTextRelocation(inFileList, FILE*, unsigned char);
unsigned long OutputDataRelocation(inFileList, FILE*);
unsigned long OutputSymbolTable(inFileList, FILE*, unsigned long*);
```

```
/* patch.c -- patch holes in object-files */

#ifndef PATCH_INCLUDED
#include "patch.h"
#endif

#ifdef __MSDOS__
#include <stdlib.h>
#else
#include <sys/stat.h>
int unlink(char *);
int fclose(FILE *);
int fflush(FILE *);
int fprintf(FILE *, char *, ...);
int fseek(FILE*, long int, int);
int fwrite(void *, int, int, FILE*);
#if 0
void free();
char *malloc(), *calloc();
int exit();
#endif
#endif

SymNode *FindSegmentName(FListNode *FLNode) {
    SymNode *snode;
    for (snode = FLNode->symbol_table.head;
         snode != NULL; snode=snode->next) {
        if (snode->entry->n_type == N_SEGMENT) {
            return snode;
        }
    }
    return NULL;
}

void UpdateSymbolOffsets(inFileList L) {
    FListNode *FLNode;
    SymNode *snode;
    int x = 0;
    char *segment_name;
    int textoffset = 0, dataoffset = 0;

    for (FLNode = L.head; FLNode != NULL; FLNode = FLNode->next) {
        if (!FLNode->inFile) continue;
        snode = FindSegmentName(FLNode);
        if (snode) {
            segment_name = snode->entry->n_un.n_name;
            x = 0;
            for (x = 0; x < L.num_segments; x++)
              if (!strcmp(L.segment_names[x], segment_name)) {
                  update_offsets(FLNode->symbol_table, textoffset,
                                 L.segment_offsets[x]);
                  L.segment_offsets[x] += FLNode->header.a_data;
                  break;
              }
            textoffset += TrueSize(FLNode);
        } else {
            update_offsets(FLNode->symbol_table, textoffset, dataoffset);
            textoffset += TrueSize(FLNode);
            dataoffset += FLNode->header.a_data;
        }
    }
}

char **GetSegmentNames(inFileList *L) {
    FListNode* FLNode;
    SymNode *snode;
    char **names = NULL;
    int y = 0;
    char found = FALSE;
    char userX = FALSE;

    for (FLNode = L->head; FLNode != NULL; FLNode = FLNode->next)
      y++;

    names = (char**)calloc(sizeof(char*),y);

    L->num_segments = 0;
    for (FLNode = L->head; FLNode != NULL; FLNode = FLNode->next) {
        if (!isMultiDataSeg(FLNode)) {
            if ((snode = FindSegmentName(FLNode)) != NULL) {
                for (found = FALSE, y = 0; y < L->num_segments; y++)
                  if (!strcmp(names[y], snode->entry->n_un.n_name)) {
                      found = TRUE;
                      break;
                  }

                if (!found) {
                    names[L->num_segments] =
                      (char*)calloc(sizeof(char),
                                    snode->entry->n_un.n_name + 1);
                    sprintf(names[L->num_segments], snode->entry->n_un.n_name);
                    snode->entry->n_value = L->num_segments * 8;
                    L->num_segments++;
                }
            } else if (!userX) {
                /* names[L->num_segments] = (char*)calloc(sizeof(char), 6);
                   sprintf(names[L->num_segments],"userX");
                   L->num_segments++;
                   */
                userX = TRUE;
            }
        } else {
            /* must search for all segment symbols... */
            for (snode = FLNode->symbol_table.head;
                 snode != NULL; snode=snode->next) {
                if (snode->entry->n_type == N_SEGMENT) {
                    for (found = FALSE, y = 0; y < L->num_segments; y++)
                      if (!strcmp(names[y], snode->entry->n_un.n_name)) {
                          found = TRUE;
                          break;
                      }

                    if (!found) {
                        names[L->num_segments] =
                          (char*)calloc(sizeof(char),
                                        snode->entry->n_un.n_name + 1);
                        sprintf(names[L->num_segments],
                                snode->entry->n_un.n_name);
                        snode->entry->n_value = L->num_segments * 8;
                        L->num_segments++;
                    }
                }
            }
        }
    }

    if (L->num_segments) {
        L->segment_offsets = (int*)calloc(sizeof(int), L->num_segments - 1);
        for (y = 0; y < L->num_segments; y++)
          L->segment_offsets[y] = 0;
```

```
    }
    return names;
}

SymNode* FindExternalSym(inFileList L,FListNode *exception, SymNode *toFind) {
    /* look for an externally-defined symbol in L (except for FListNode) with
       the name of s */
    FListNode *N;
    SymNode *snode = NULL;

    for (N = L.head; N != NULL; N = N->next) {
        /* if (N == exception) continue; */
        for (snode = N->symbol_table.head; snode != NULL; snode=snode->next) {
            if (!strcmp(snode->entry->n_un.n_name,
                        toFind->entry->n_un.n_name) &&
                (isSegmentSymbol(*snode->entry) ||
                 (isExternalSymbol(*snode->entry) &&
                  !isUndefSymbol(*snode->entry)))) {
                return snode;
            }
        }
    }

    return NULL;
}

void update_offsets(SymList L, long int toff, long int doff) {
    SymNode *snode;

    for (snode = L.head; snode != NULL; snode = snode->next) {
        if (isTextSymbol(*snode->entry))
          snode->entry->n_value += toff;
        if (isDataSymbol(*snode->entry))
          snode->entry->n_value += doff;
    }
}

int patch_symbols(SymList S, inFileList L, FListNode *exception
                  /* SymList *duplicates, SymList *undefined */) {
    /* look for undefined symbols...*/
    int error = 0;
    SymNode *snode, *newnode;

    for (snode = S.head; snode != NULL; snode = snode->next) {
        if (isUndefSymbol(*snode->entry))
          if ((newnode = FindExternalSym(L, exception, snode)) != NULL) {
              /* found a valid symbol to patch the current table entry...*/
              snode->entry->n_value = newnode->entry->n_value;
              if (newnode->entry->n_type == N_SEGMENT)
                snode->entry->n_type  = N_SEGMENT;
              else
                if (newnode->entry->n_type & N_DATA)
                  snode->entry->n_type = N_DATA | N_EXT;
                else if (newnode->entry->n_type & N_TEXT)
                  snode->entry->n_type = N_TEXT | N_EXT;
                else
                  snode->entry->n_type  = N_ABS | N_EXT; /* ???
                                                            Change this to
                                                            reflect real types
                                                          of symbols */
          } else error = -1;
    }
    return error;
}

int check_twoscomplement_range(long number, int bits) {
    /* returns nonzero if number could be represented in bits number of bits
       (two's complement notation.)  Returns zero otherwise. */
    long lower_range, upper_range;
    lower_range = -(1 << (bits-1));
    upper_range = (1 << (bits-1)) -1;
    if ((number < lower_range) || (number > upper_range)) return 0;
    else return 1;
}

int check_normal_range(unsigned long number, int bits) {
    /* returns nonzero number could be represented in bits number of bits
       (binary notation.)  Returns zero otherwise. */
    unsigned long lower_range, upper_range;
    lower_range = 0;
    upper_range = (1 << bits) -1;
    if ((number < lower_range) || (number > upper_range)) return 0;
    else return 1;
}

int patch_operation(codebits *C, unsigned long op_offset,
                    unsigned long value, int type) {
    long int sV;

    switch(type) {
    case ABS_AECI_RELOC:
        if (!check_twoscomplement_range(value,7)) {
            fprintf(stderr,"\nCannot patch %x into 7-bit hole.\n", value);
            return 0;
        }
        value = (value & ~CLASS_AECI_MASK);
        *C &= CLASS_AECI_MASK;
        *C |= value;
        break;
    case ABS_B_RELOC:
        if (!check_twoscomplement_range(value,11)) {
            fprintf(stderr,"\nCannot patch %x into 11-bit hole.\n",
                    value);
            return 0;
        }
        value = (value & ~CLASS_B_MASK);
        *C &= CLASS_B_MASK;
        *C |= value;
        break;
    case ABS_G_RELOC:
        if (!check_twoscomplement_range(value,18)) {
            fprintf(stderr,"\nCannot patch %x into 18-bit hole.\n",
                    value);
            return 0;
        }
        value = (value & ~CLASS_G_MASK);
        *C &= CLASS_G_MASK;
        *C |= value;
        break;
    case REL_G_RELOC:
        sV = value - op_offset;
        if (!check_twoscomplement_range(sV,18)) {
            fprintf(stderr,"\nCannot patch %x into 18-bit hole.\n",
                    sV);
            return 0;
        }
        value = (sV & ~CLASS_G_MASK);
        *C &= CLASS_G_MASK;
        *C |= value;
        break;
```

```c
    default:
        break;
    }
    return 1;
}

void patch_data(char *data, unsigned long value, int type) {
    ULong64 u64;

    switch(type) {
    case DATA_I64_RELOC:
        u64 = value;
        memcpy(data,&u64,sizeof(ULong64));
        break;
    default:
        break;
    }
}

int patchHoles(FListNode *FLNode,
               unsigned long prev_text_off,
               unsigned long prev_data_off,
               unsigned char verbosemode) {

    RelocNode *rnode;
    SymNode   *snode;
    codebits *CPtr;
    char *dataPtr;

    for (rnode = FLNode->relocation.head; rnode != NULL; rnode = rnode->next) {

        /* get a pointer to the instruction which needs to be patched */
        if (rnode->entry->r_address % 8) {
            CPtr = &(FLNode->TextSegment[(rnode->entry->r_address -
                                          (rnode->entry->r_address % 8)) / 4]);
        } else {
            CPtr = &(FLNode->TextSegment[rnode->entry->r_address / 4 + 1]);
        }

        if (rnode->entry->r_extern) {
            /* an external symbol is referenced. */
            if ((snode = FindSymTabEntry(FLNode->symbol_table,
                                         rnode->entry->r_index)) == NULL) {
                /* could not find symbol */
                fprintf(stderr,
                        "lnk: could not find symbol to patch address %d\n"
                        "please save the command used and the object files "
                        "and report this error.\n",
                        rnode->entry->r_address);
                return 0;
            } else if (isUndefSymbol(*snode->entry)) {
                fprintf(stderr,
                        "lnk: found an undefined symbol to patch "
                        "address %d\nplease save the command used and "
                        "the object files\ninvolved and report "
                        "this error.\n", rnode->entry->r_address);
                return 0;
            } else {
                if (verbosemode)
                  fprintf(stdout,"patching external $%08X: %X --> ",
                          rnode->entry->r_address + prev_text_off,
                          Bits2Long(*CPtr));
                patch_operation(CPtr,
                                rnode->entry->r_address + prev_text_off,
                                snode->entry->n_value,
                                rnode->entry->r_type);
                if (verbosemode)
                  fprintf(stdout,"%X\n",Bits2Long(*CPtr));
            }
        } else if (rnode->entry->r_index == 0xffffff) {
            /* don't patch, because this is special data relocation */

        } else {
            if ((snode = FindSymTabEntry(FLNode->symbol_table,
                                         rnode->entry->r_index)) == NULL) {
                fprintf(stderr,
                        "lnk: could not find symbol to patch address %d\n"
                        "please save the command used and the object files "
                        "and report this error.\n",
                        rnode->entry->r_address);
                return 0;
            } else if (isUndefSymbol(*snode->entry)) {
                fprintf(stderr,
                        "lnk:  non-external symbol is undefined!\n"
                        "please save the command line used and the\n"
                        "involved object files and report this error.\n");
                return 0;
            } else if ((!isAbsSymbol(*snode->entry)) &&
                       (!isSegmentSymbol(*snode->entry))) {
                if (verbosemode)
                  fprintf(stdout,"patching local $%08X: %X --> ",
                          rnode->entry->r_address + prev_text_off,
                          Bits2Long(*CPtr));
                patch_operation(CPtr,
                                rnode->entry->r_address + prev_text_off,
                                snode->entry->n_value,
                                rnode->entry->r_type);
                if (verbosemode)
                  fprintf(stdout,"%X\n",Bits2Long(*CPtr));
            }
        }
    }

    for (rnode = FLNode->datareloc.head; rnode != NULL; rnode = rnode->next) {
        dataPtr = &(FLNode->DataSegment[rnode->entry->r_address]);

        if (rnode->entry->r_extern) {
            /* an external symbol is referenced. */
            if ((snode = FindSymTabEntry(FLNode->symbol_table,
                                         rnode->entry->r_index)) == NULL) {
                fprintf(stderr,
                        "lnk: could not find symbol to patch address %d\n"
                        "please save the command used and the object files "
                        "and report this error.\n",
                        rnode->entry->r_address);
                return 0;
            } else if (isUndefSymbol(*snode->entry)) {
                fprintf(stderr,
                        "lnk: found an undefined symbol to patch "
                        "address %d\nplease save the command used and "
                        "the object files\ninvolved and report "
                        "this error.\n", rnode->entry->r_address);
                return 0;
            } else {
                if (verbosemode)
                  fprintf(stdout,"patching data external $%08X: %X --> ",
                          rnode->entry->r_address + prev_data_off,
                          *(unsigned long*)dataPtr);
                patch_data(dataPtr,
                           snode->entry->n_value,
```

```
                          rnode->entry->r_type);
                if (verbosemode)
                  fprintf(stdout,"%X\n", *(unsigned long*)dataPtr);
            }
        } else if (rnode->entry->r_index == 0xffffff) {
            /* don't patch */
        } else {
            if ((snode = FindSymTabEntry(FLNode->symbol_table,
                                         rnode->entry->r_index)) == NULL) {
                fprintf(stderr,
                        "lnk: could not find symbol to patch address %d\n"
                        "please save the command used and the object files "
                        "and report this error.\n",
                        rnode->entry->r_address);
                return 0;
            } else if (isUndefSymbol(*snode->entry)) {
                fprintf(stderr,
                        "lnk: found an undefined local symbol to patch "
                        "address %d\nplease save the command used and "
                        "the object files\ninvolved and report "
                        "this error.\n", rnode->entry->r_address);
                return 0;
            } else if ((!isAbsSymbol(*snode->entry)) &&
                       (!isSegmentSymbol(*snode->entry))) {
                if (verbosemode)
                  fprintf(stdout,"patching data local $%08X: %X --> ",
                          rnode->entry->r_address + prev_data_off,
                          *(unsigned long*)dataPtr);
                patch_data(dataPtr,
                           snode->entry->n_value,
                           rnode->entry->r_type);
                if (verbosemode)
                  fprintf(stdout,"%X\n", *(unsigned long*)dataPtr);
            }
        }
    }
    return 1;
}

int CheckSymbolTables(inFileList FList, SymList *duplist, SymList *undeflist) {
    unsigned long a_syms = 0, i;
    SymNode *snode, *tempnode;
    FListNode *FLNode, *FLNode2;
    char **sym_names, found = FALSE;
    long dup_symbol = -1;
    int error = FALSE;

    /* allocate an array of symbols-name strings */
    sym_names = (char**)calloc(TotalSymbols(FList), sizeof(char*));

    for (FLNode = FList.head; FLNode != NULL; FLNode = FLNode->next) {
        for (snode = FLNode->symbol_table.head;
             snode != NULL;
             snode = snode->next) {
            if (isUndefSymbol(*(snode->entry))) {
                /* find a defined symbol in filelist */
                found = FALSE;
                for (FLNode2 = FList.head;
                     FLNode2 != NULL;
                     FLNode2 = FLNode2->next) {
                    if (FLNode2 != FLNode) {
                        for (tempnode = FLNode2->symbol_table.head;
                             tempnode != NULL;
                             tempnode = tempnode->next) {
                            if (!strcmp(snode->entry->n_un.n_name,
                                        tempnode->entry->n_un.n_name) &&
                                (isSegmentSymbol(*(tempnode->entry)) ||
                                 (isExternalSymbol(*(tempnode->entry)) &&
                                  !isUndefSymbol(*(tempnode->entry)))))
                              found = TRUE;
                        }
                    } else {
                        for (tempnode = FLNode2->symbol_table.head;
                             tempnode != NULL;
                             tempnode = tempnode->next) {
                            if (!strcmp(snode->entry->n_un.n_name,
                                        tempnode->entry->n_un.n_name) &&
                                (isSegmentSymbol(*(tempnode->entry))))
                              found = TRUE;
                        }
                    }
                }

                if (found == FALSE) {
                    if ((tempnode = SymNode_create()) != NULL)
                      tempnode->entry = snode->entry;
                    SymList_add(undeflist,tempnode);
                    error = TRUE;
                }
            }
        }
    }

    for (FLNode = FList.head; FLNode != NULL; FLNode = FLNode->next) {
        for (snode = FLNode->symbol_table.head;
             snode != NULL;
             snode = snode->next) {
            if (isExternalSymbol(*(snode->entry)) &&
                !isUndefSymbol(*(snode->entry))) {
                for (i = 0, dup_symbol = -1; i < a_syms; i++)
                  if (!strcmp(sym_names[i], snode->entry->n_un.n_name)) {
                      /* found an existing symbol already in the
                         symbol table */
                      dup_symbol = i;
                      break;
                  }

                if (dup_symbol == -1) {
                    sym_names[a_syms] = snode->entry->n_un.n_name;
                    a_syms++;
                } else {
                    /* place in duplicate list */
                    if ((tempnode = SymNode_create()) != NULL)
                      tempnode->entry = snode->entry;
                    SymList_add(duplist,tempnode);
                    error = TRUE;
                }
            }
        }
    }
    free(sym_names);
    return error;
}
```

```
/* patch.h -- patch object files */

#define PATCH_INCLUDED

#ifndef DATASTRUCT_INCLUDED
#include "datastruct.h"
#endif

int    patchHoles(FListNode*, unsigned long, unsigned long, unsigned char);
int    patch_symbols(SymList, inFileList, FListNode*);
void   update_offsets(SymList, long int, long int);
int    CheckSymbolTables(inFileList FList,
                         SymList *duplist,
                         SymList *undeflist);
char** GetSegmentNames(inFileList*);
void   UpdateSymbolOffsets(inFileList);
```