Texas Instruments DSP Dynamic Run-Time Loader
by
Morris M. Tao

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology
May 23, 2001
Copyright 2001 Morris M. Tao. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author__________________________
Department of Electrical Engineering and Computer Science
May 23, 2001

Certified by__________________________
Jon Rowlands
VI-A Company Thesis Supervisor, Texas Instruments

Certified by__________________________
Martin Rinard
M.I.T. Thesis Supervisor

Accepted by__________________________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Texas Instruments DSP Dynamic Run-Time Loader
by
Morris M. Tao
Submitted to the
Department of Electrical Engineering and Computer Science
May 23, 2001
In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

The increased demand in applications of Digital Signal Processors (DSP) are reaching the limits of the basic method of statically loading libraries. These statically loaded libraries are fixed to the location in memory where they can be executed. Therefore these inflexible libraries are difficult to use in multitasking environments, are not portable between different DSP architectures and are an inefficient use of RAM. A Dynamic Run-Time Loader for libraries adds an abstraction layer between the DSP kernel and the library. This solution will remove the memory inflexibility of the static model while improving code reuse, portability and multitasking ability, to help meet the demands of tomorrows DSP applications.

Thesis Supervisor: Martin Rinard
Title: Assistant Professor, Department of Electrical Engineering and Computer Science
Acknowledgements

I would like to thank the following individuals and organizations for their help and support which made my work possible.

?? My parents for their unique blend of guidance, encouragement and tough love.

?? Jon Rowlands for his leadership, advice and patience. This project would not have been possible without his breadth of knowledge and the willingness to impart some of it onto me.

?? Texas Instruments and the VI-A coop program at MIT which provided me with the wonderful opportunity and environment to pursue this project.

?? Helen Chuah for her patience and support during my work.

?? My thesis advisor, Prof. Martin Rinard for his suggestions and help throughout my thesis writing.
# Table of Contents

1 PURPOSE .......................................................................................................................... 10
   1.1 CURRENT ISSUES .................................................................................................. 10
   1.2 DEVELOPMENT HARDWARE ............................................................................. 10
   1.3 CURRENT LIBRARY ARCHITECTURE .................................................................. 12
   1.4 NEW DEMANDS .................................................................................................... 13
   1.5 GRANULARITY OF CODE REPLACEMENT .......................................................... 14
   1.6 CODE REUSE ...................................................................................................... 15
      1.6.1 Portability between DSPs .............................................................................. 15
      1.6.2 Code Duplication ......................................................................................... 16
   1.7 KERNEL SECURITY ............................................................................................... 16
   1.8 MEMORY INFLEXIBILITY ...................................................................................... 16

2 DYNAMIC LOADER ....................................................................................................... 18
   2.1 DYNAMIC LOADING PROCESS .......................................................................... 18
   2.2 COFF .................................................................................................................... 19
      2.2.1 COFF Header ............................................................................................... 20
      2.2.2 COFF Sections ............................................................................................ 20
   2.3 RELOCATION INFORMATION .............................................................................. 21
   2.4 REGULAR RELOCATIONS ................................................................................... 22
   2.5 ARITHMETIC RELOCATIONS ............................................................................. 22
   2.6 OTHER TABLES .................................................................................................... 22

3 DYNAMIC LOADING ALGORITHM ............................................................................... 24
   3.1 FINDING LIBRARY ............................................................................................... 24
   3.2 LOADING LIBRARY HEADERS .......................................................................... 24
   3.3 MEMORY ALLOCATION ....................................................................................... 24
   3.4 LOOP THROUGH SECTIONS ............................................................................ 25
      3.4.1 Data Section Relocation .............................................................................. 25
      3.4.2 Code Section Relocation ............................................................................ 26
   3.5 LIBRARY ENTRYPONT ......................................................................................... 27
   3.6 FREEING LIBRARY ............................................................................................. 27

4 EXPECTATIONS ............................................................................................................... 28
   4.1 ABSTRACTION LAYER ....................................................................................... 28
   4.2 MULTITASKING ................................................................................................... 29
   4.3 CODE REUSE ...................................................................................................... 29
      4.3.1 Field Upgrades ............................................................................................ 29
      4.3.2 Code Portability ......................................................................................... 30
      4.3.3 Code Overlay ............................................................................................ 30
   4.4 KERNEL SECURITY ............................................................................................... 30

5 TRADEOFFS .................................................................................................................. 31
   5.1 ADDED COMPLEXITY ......................................................................................... 31
Appendix A
Dynamic Loader Usage Documentation

1 INTRODUCTION .............................................................................................................. 42
1.1 WHY A DYNAMIC LOADER FOR DSPS ................................................................. 42
1.2 SOLUTION .................................................................................................................. 42
1.3 CAPABILITIES .......................................................................................................... 42
1.4 LIMITATIONS .......................................................................................................... 43

2 FILE STRUCTURE ......................................................................................................... 44
2.1 FILES ......................................................................................................................... 44
   2.1.1 File Dependencies .......................................................................................... 44
2.2 CLASSES ..................................................................................................................... 45
   2.2.1 Dynamic Loader Class .................................................................................. 45
   2.2.2 File Class ...................................................................................................... 45
   2.2.3 Memory Class ............................................................................................... 45
2.3 XDAIS ......................................................................................................................... 45

3 USAGE OF DYNAMIC LOADER ............................................................................... 46
3.1 PREPARING DYNAMIC LOADER ........................................................................... 46
   3.1.1 Add Dynamic loader files into application .................................................... 46
   3.1.2 Update DLLMem, DLLFile based on platform ............................................. 46
   3.1.3 Customize DSP/BIOS memory constraints .................................................. 48
   3.1.4 Update Memory Constraints constants in Dynamic Loader files .............. 50
3.2 PREPARING DYNAMIC LIBRARY ......................................................................... 50
   3.2.1 Writing Algorithm ....................................................................................... 50
   3.2.2 Library Generation ...................................................................................... 51
   3.2.3 Library Wrapper ......................................................................................... 51
3.3 RUNNING THE DYNAMIC LOADER ....................................................................... 52
   3.3.1 Instantiating XDAIS objects (DLLMEM, DLLFILE, DLL) ....................... 53
   3.3.2 Locate Library .............................................................................................. 54
   3.3.3 Relocating Library ...................................................................................... 55
   3.3.4 "Calling" Library to open wrapper ............................................................... 55
   3.3.5 Using the Relocated Library ..................................................................... 55
   3.3.6 Freeing Library .......................................................................................... 55

4 DYNAMIC LOADING ALGORITHM ............................................................................. 57
4.1 USING THE DYNAMIC LOADER .......................................................................... 57
   4.1.1 DLLFILE Definition ............................................................................... 57
   4.1.2 Accessing DLLFILE ................................................................................ 58
   4.1.3 Benefits ....................................................................................................... 58
   4.1.4 DLLMEM Definition ................................................................................ 58
   4.1.5 DLL Definition .......................................................................................... 58
   4.1.6 Finding the Library .................................................................................... 59
   4.1.7 Running the Library .................................................................................. 59
4.2 .CINIT ................................................................................................................. 60

5 CODE DESCRIPTION .......................................................................................... 61

5.1 DLLOBJ_IA.H .................................................................................................. 61
  5.1.1 DLL_MemRec ........................................................................................... 61
  5.1.2 DLL_Obj ................................................................................................... 62
  5.1.3 DLL_Handle ............................................................................................ 62
  5.1.4 DLL_Params ............................................................................................. 62
  5.1.5 DLL_Fxns ............................................................................................... 63

5.2 DLL.H ............................................................................................................... 64
  5.2.1 DLL_Load_file_header ............................................................................. 64
  5.2.2 DLL_Num_alloc ........................................................................................ 64
  5.2.3 DLL_Alloc ................................................................................................. 64
  5.2.4 DLL_Alloc_Memory ............................................................................... 65
  5.2.5 DLL_Free .................................................................................................. 65
  5.2.6 DLL_Create .............................................................................................. 65
  5.2.7 Load_data ................................................................................................. 66
  5.2.8 Load_Relocate_section ........................................................................... 66
  5.2.9 Load_Relocate .......................................................................................... 67
  5.2.10 Cinit_data .............................................................................................. 67
  5.2.11 DLL_Init .................................................................................................. 67

5.3 DLLRELOCATE.H .......................................................................................... 68
  5.3.1 Reloc_vars ................................................................................................. 68
  5.3.2 Read_Reloc ............................................................................................... 68
  5.3.3 Read_symbol ............................................................................................. 68
  5.3.4 relocate ...................................................................................................... 69
  5.3.5 Relocate_data ............................................................................................ 69

5.4 DLLMEM.H ..................................................................................................... 70
  5.4.1 Constants ................................................................................................... 70
  5.4.2 DLLMEM_Obj .......................................................................................... 70
  5.4.3 DLLMEM_Params .................................................................................... 70
  5.4.4 DLLMEM_Fxns ........................................................................................ 70
  5.4.5 DLLMEM_Allocate_1 ............................................................................... 71
  5.4.6 DLLMEM_Copy_data_to_program ......................................................... 71
  5.4.7 DLLMEM_Write_char_1 .......................................................................... 71
  5.4.8 DLLMEM_Read_char_1 ........................................................................... 72
  5.4.9 DLLMEM_Write_int_1 ............................................................................. 72
  5.4.10 DLLMEM_Read_int_1 ............................................................................ 72
  5.4.11 DLLMEM_Write_long_1 ........................................................................ 73
  5.4.12 DLLMEM_Read_long_1 .......................................................................... 73
  5.4.13 DLLMEM_Write_string_1 ....................................................................... 73
  5.4.14 DLLMEM_Read_string_1 ....................................................................... 74
  5.4.15 DLLMEM_Free_1 .................................................................................. 74

5.5 DLLFILE.H ....................................................................................................... 74
  5.5.1 DLLFILE_Obj ........................................................................................... 74
  5.5.2 DLLFILE_Params .................................................................................... 75
  5.5.3 DLLFILE_Fxns ........................................................................................ 75
5.5.4  DLLFILE.Seek_1................................................................. 75
5.5.5  DLLFILE.Tell_1................................................................. 75
5.5.6  DLLFILE.Get_char_1.......................................................... 76
5.5.7  DLLFILE.Get_short_1......................................................... 76
5.5.8  DLLFILE.Get_long_1.......................................................... 76
5.5.9  DLLFILE.Get_string_1......................................................... 77

6  LIMITATIONS................................................................................. 78

6.1  ERROR CHECKING...................................................................... 78
  6.1.1  COFF Validation................................................................. 78
6.2  MEMORY ALLOCATION............................................................ 78
  6.2.1  Program Space................................................................. 79
6.3  NEAR MODE............................................................................ 80
6.4  SHARED DATA/PROGRAM MEMORY........................................ 80
6.5  PARTIAL LOADING LIMITATIONS........................................... 80
6.6  FIXED PROGRAM MEMORY RELOCATION BUFFER................... 81

7  WARNINGS.................................................................................. 82

7.1  ARITHMETIC OPERATIONS WITHIN ADDRESS REFERENCES IN ASSEMBLY 82
7.2  FAR MODE............................................................................. 82

8  NOTES.......................................................................................... 83

8.1.................................................................................................. 83
8.2.................................................................................................. 83
8.3.................................................................................................. 83
8.4.................................................................................................. 83
8.5.................................................................................................. 83
8.6.................................................................................................. 83
Table of Figures

Figure 1 - DSP based MP3 Player Block Diagram (Copyright Texas Instruments) ........ 11
Figure 2 - Static Library Load. Requested applications in Library B from secondary
    memory are loaded into a fixed location in RAM overwriting Library A. .............. 13
Figure 3 - Dynamic Load. Library B, minus the kernel, is loaded anywhere into RAM. 19
Figure 4 - COFF File Structure (Copyright Texas Instruments)............................... 21
1 Purpose

1.1 Current Issues

The real time processing of data by Digital Signal Processors or DSPs make them useful in cell phones, the ABS system in your car, MP3 players and a host of other real-time sensitive applications. When newer embedded applications with more features and capabilities are developed it is preferred to use older DSPs to save on the costs of using newer, faster DSPs. The process of squeezing increasingly demanding applications into older DSPs strains both the processing and memory resources of these chips. While optimizations in the code can made to address speed concerns, other large hindrances are in DSP memory management, code reuse and multitasking.

1.2 Development Hardware

Before examining the issues further, it will be helpful to gain familiarity with the hardware used in this project to better understand the context of the problem.

In DSP devices such as audio (MP3) players, a kernel resides in main memory (RAM) on the DSP. Because of the cost of RAM, devices are complemented with a more abundant supply of slower secondary memory in the form of a ROM, compact flash card, or similar memory separate from the DSP. Executable code usually resides permanently in this slower secondary memory. This code contains functions and data, referred henceforth as a library.

The actual development hardware used in this project was a TI DSP based audio board designed by the Internet Audio group of Texas Instruments. The board, which is
named the “Leia” is based on a TI C5416 DSP with 128K words of RAM, LCD screen, support for Compact Flash and SmartMedia formats and a headphone jack. A 30 MB Compact Flash card was used during this development. Also available is a JTAG connection to interface the Leia with DSP development software on a PC. The Leia board is used as a platform for developing audio players and associated software. A block diagram of a similar TI development board is shown in Figure 1.1.

The Dynamic Loader was entirely developed and tested on the Leia board, but it was designed to be portable to many different TI DSP based systems. Since this project was completed in the Internet Audio Group of TI, emphasis in this document is placed on audio applications, although the Dynamic Loader can be used in many other TI DSP applications.

Figure 1 - DSP based MP3 Player Block Diagram (Copyright Texas Instruments)
1.3 Current Library Architecture

Each of the libraries resident on secondary memory contains the code and data needed for a specific task, such as decoding an MP3 file. Each library also contains its own copy of the kernel. Although the kernel code between libraries is essentially the same, it is compiled with the library because of specific static references between kernel and the code. (i.e. the location of functions and data structures within this specific library).

These libraries are linked statically meaning the library is linked assuming it will be loaded into one specific address in RAM when it is used. All of the address references, such as branches, jumps or data references within this code are thus linked to fixed addresses. Consequently, the library will not function correctly if it is loaded into another address. For more information on static libraries and linking see “Linkers and Loaders” listed in the bibliography.

When a new task or application is requested, and the library associated with this new operation is not already in RAM, the kernel searches secondary memory for the needed library. When found, the new library is then copied from secondary memory to a fixed location in RAM. This new library (containing a copy of the kernel) overwrites the previous copy of the kernel and library. The system is then reset, automatically executing the new library. See Figure 1.1.

These statically linked libraries are sufficient for current needs in DSP embedded systems. Although DSPs are very versatile, in any given DSP system it usually performs a given set of tasks over and over, which means those tasks can be compiled into one library that is statically linked to a specific address. Since libraries are statically loaded,
when a new library is needed, it can be copied straight from secondary memory to RAM and used immediately. Problems with this current method arise when DSP systems are faced with the demand of running more than one library in memory at once.

![Static Library Load Diagram](image)

Figure 2 - Static Library Load. Requested applications in Library B from secondary memory are loaded into a fixed location in RAM overwriting Library A.

### 1.4 New Demands

One example of where new demands are straining the current static library architecture is in audio devices such as the Leia board. When new audio formats such as
MP3, Windows Media, AC-3, etc, are to be supported in an audio player, a new library needs to be added for each format to be supported (some decoders take up all available RAM). In addition, extra functions such as a software graphic equalizer, distortion or special effects can also be placed into separate libraries. The benefit would be the ability to use combinations of these libraries to create highly customizable environments. For example, one might want to play an MP3 song and load up a graphic equalizer library, then later unload the graphic equalizer and load in a distortion library without have the song interrupted.

Despite these benefits, a number of difficulties arise when trying to load, unload and use multiple libraries simultaneously, as in this example, using static libraries. These limitations are described below.

1.5 Granularity of Code Replacement

The coarse granularity of exchanging libraries is the biggest problem with static libraries. When replacing a static library, the entire memory is overwritten with the new library. Since all of RAM is affected, this prevents other libraries from being loaded.

Since the library might not utilize all of the memory it’s loaded into, a potential solution is to partition memory into smaller, independently loadable regions to let other libraries use that space at the same time. In this scenario library A can be loaded into one partition and library B can be loaded into another partition within memory. The problem is that since these libraries are statically linked, they are fixed to one specific region. Two libraries that are linked to the same address, must be loaded to the same partition, severely limiting the flexibility of the system. The partition also limits how much
memory an application can use and creates inefficiencies if an application does not need all of the memory in the partition. Too large a partition and there is a waste of unused memory in a partition and lowers the number of libraries that can be loaded simultaneously. Too few partitions and applications might not be able to fit into memory.

The inherent inefficiencies with this proposal are due to the statically linked libraries in this architecture.

1.6 Code Reuse

Another inefficiency lies in lack of code reuse. As stated earlier, libraries are “hard-coded” to a copy of the kernel for simplicity. The tradeoff is that this creates a very memory-rigid set of libraries and an inefficient use of secondary memory.

1.6.1 Portability between DSPs

When libraries that have been written for a specific DSP chip are ported to another DSP model with the same instruction set, it cannot be simply copied over to operate correctly. This is because the two DSPs might address memory differently or have completely different memory architectures or sizes. In order to share libraries, the source needs to be re-linked into a new library in order to execute correctly on the different DSP. This is another limitation of statically linked libraries which prevents code generated on one DSP to be easily used on another. Ideally, since the two DSPs share instruction sets, libraries should be able to be interchanged between two DSPs without having to re-link the library.
1.6.2 Code Duplication

A very apparent inefficiency with the current method is in the waste of space on the secondary memory. Since the copy of the kernel in each library is nearly an exact copy of each other, much of the data in secondary memory is duplicated many times over.

1.7 Kernel Security

The copies of the kernel included with the static libraries on the secondary memory are insecure. Since third parties may develop libraries for the audio device, they need to have access to the kernel source since it must be compiled with a new library. Those unfamiliar with the kernel can unintentionally introduce bugs and errors into the source when creating a new library. More importantly, a malicious programmer would be able to steal the kernel binary or even modify it to do damage to the other libraries since the kernel exists on secondary memory.

1.8 Memory Inflexibility

One last important limitation of statically linked libraries that has already been mentioned is the inflexibility of memory usage. Since the compiled kernel and libraries are hard coded in their addresses, they can only be loaded into a specific memory location in order to work correctly. If the library was loaded into a different memory address, the code would not run. Clearly, this requirement severely limits the memory flexibility demanded by multitasking environments. It cannot be assumed that the range of memory library required by a new library is unoccupied.
2 Dynamic Loader

A need exists for a DSP run time loader which can dynamically load libraries into RAM when needed. The loader will reside with the kernel in RAM at all times. The libraries will still be found on secondary memory, but without copies of the kernel. These libraries will be in a format (described in section 2.2) which will contain additional information about their specific memory needs. When the code for a feature not currently available in the library in RAM is requested, (such as the decoding of an audio format different from the current one) the kernel communicates this request to the Dynamic Loader.

2.1 Dynamic Loading Process

The role of the Dynamic Loader is to first locate the requested library within the secondary memory. When the library is found, the Dynamic Loader then analyzes the memory requirements of the library to ensure there are sufficient resources. If there are enough resources, the loader will then allocate RAM for the program code and data in the library. The code and data is then copied or loaded into RAM from secondary memory. Lastly the loader must adapt the library to run correctly at its loaded address. This process, called relocation, will fix up all the address references within the code and data to reflect their new locations in memory. (See section 2.3). When relocations have finished, the loader updates the kernel references with pointers to the functions and data of the newly loaded library. Lastly the requested function from the library is called from RAM. See Figure 3 for the new architecture. See Section 3 for a more detailed
description of this algorithm and Appendix A, Section 4 for specific examples on how to use the Dynamic Loader.

![Diagram of Dynamic Load](image)

Figure 3 - Dynamic Load. Library B, minus the kernel, is loaded anywhere into RAM.

### 2.2 COFF

Each library resides in secondary memory in a specific file format, the Common Object File Format or COFF. It is a common format which is output by many compilers after the compiling and liking of code and data. It is used as an interim medium for code and data before it is loaded into RAM to be executed. Because this is exactly the purpose of the library, to be stored on secondary memory until loaded into RAM, and the TI compiler already outputs linked code in COFF, this was the best choice for the library file
format. See Figure 4 for a diagram of the COFF format. More information on the COFF format is found in Appendix A, Section 8.1.

2.2.1 COFF Header

At the beginning of each COFF file is a file header which contains information regarding the library. Such as: the type of COFF this file is, the total number of sections within this library, locations of certain parts of the COFF, etc.

2.2.2 COFF Sections

The COFF file packages the code and data into similar contiguous pieces, called sections. There are usually code, initialized data and uninitialized data sections in most libraries. Each section contains one specific type of information. For example a library might consist of one section of code and two sections of data.

Each section also contains the metadata about that section such as size, alignment and references to symbols and location of relocation entries (described in section 2.3). This metadata is stored in a section header. All of the section headers are located in the COFF after the file header and before the beginning of the actual sections.
2.3 Relocation Information

The relocation information is used to identify the destination of branch instructions and other memory address references. Relocation information is used by loaders to “fix up” these addresses when the sections they refer to are allocated into different locations relative to its own section. For example, a section might start at address 0. Within that section might be a branch instruction to address 100, which happens to still be within the section. If this section were to now be moved so that it
starts at location 5, this offset must be reflected in all memory references. The new address should now be 105. The relocation process is complicated by the fact that many of these sections reference not only their internal sections, but also to external sections which can move independently.

A relocation entry exists in the COFF file for each address that needs to be relocated. There are two main types of relocation entries, regular and arithmetic.

2.4 Regular Relocations

A regular relocation entries exist for each memory reference in relocatable code or data. Each entry contains information about the expected address of the reference, pointers to symbols used in the relocation (such as Branch to “done”, where “done” is a symbol) and special flags describing the type of data or code this entry refers to. This information is used to relocate an address that refers to either the same (internal) section or a different (external) section.

2.5 Arithmetic Relocations

Arithmetic relocation entries deal with references which must be calculated through a series of arithmetic operations. Each of these relocation entries can be either an address, a constant or an arithmetic operation (+,-,*,/, AND, OR, etc). Multiple entries are used to encode a calculation for a relocation.

2.6 Other tables
Located at the end of the COFF file are a number of other tables. None of these remaining table are used in this implementation of the Dynamic Loader.

Line number entries for each section are used to map symbols back to the original source code, which is useful for debugging.

COFF files contain a symbol table, which associates textual names called “symbols” to the numeric values of addresses within the code and data. In general the symbol table may also contain symbols with no numeric value defined, which it expects to be defined in another library.

The string table is used to store names of symbols that could not fit in the symbol table.
3 Dynamic Loading Algorithm

The process of the Dynamic Loading Algorithm is explained in more detail here. Some specific implementation detail has been excluded here but can be found in the Appendix A, Section 4.

3.1 Finding Library

First, the Dynamic Loader must be told which library needs to be loaded. This responsibility is given to the kernel which decides which library to load depending on the user’s commands. The filename of the requested library is passed to the Dynamic Loader which searches within a specific location in secondary memory where libraries are stored.

3.2 Loading Library Headers

When the library has been found, the Dynamic Loader proceeds to read in the file headers of the COFF. These file headers contain information that can be used to verify that the file is a valid COFF and also provide information on the number of sections within the COFF library and the memory requirements of these sections. In addition, the file headers provide the location of relocation information within the library for each section; this information will be used during the actual relocation process.

3.3 Memory Allocation

Depending on which of these sections are to be loaded and relocated, the Dynamic Loader then allocates space in the appropriate locations in memory. Since TI DSPs have
distinct program and data spaces within RAM, the Dynamic Loader must allocate the sections within the library that pertain to either code or data. In addition, many data sections need to be memory aligned within RAM. Note at this point, no actual code or data has been loaded from the library to RAM, only the allocation of space has been completed.

The type, location, sizes of the allocated sections and location of relocation information for each section is stored in an array. This information will be used when the actual code or data for each section is loaded and relocated.

Also a small buffer is relocated within data space to be used when performing relocations in code. Further explanation in Section 3.4.2.

### 3.4 Loop Through Sections

The Dynamic Loader then loops through the array containing each section’s memory and relocation information. The code or data from each section is then loaded into memory and all of the address references relocated to fit the new memory locations of each section.

The algorithm for relocating data sections and code sections are below.

#### 3.4.1 Data Section Relocation

When a data section is to be relocated, all of the data is copied from the library to the newly allocated location in data space of RAM. At this stage all address references of the newly copied data may be incorrect since the location of the this section as well as other
sections might not be where they were assumed to be when the library was compiled. Therefore all of the address references in data need to be relocated.

The algorithm needs relocation information for each address reference so the Dynamic Loader loops through each address reference within this section, looking up the relocation information pertaining to this address reference.

The relocation information for an address reference tells the Dynamic Loader where the address reference is within this data section, what section within the library this reference actually refers to and where within the section is the code or data being referred to. For example a data reference in an array “a[1]” within a section might contain in the relocation information that this instruction is the fourth word within this section, that the actual data at the address of “a[1]” is actually found in another section and that it occurs at the tenth word within this other section.

By using this information along with the array containing search section’s memory and relocation information, the address reference can be updated to the correct final value.

3.4.2 Code Section Relocation

The difference with code is that it is traditionally not edited within a DSP, therefore the commands for editing code in the DSP do not exist. This makes it very difficult to change code once it has been copied into code space in RAM. Since the relocation process of code requires extensive editing of this code, a buffer in data memory was used to store code while it was being updated before copying it to it’s final location in code space in RAM.
Since the buffer is generally much smaller than the code that was being relocated, sequential blocks of code would need to be relocated at a time in order to fully relocate the whole code. The data buffer would be filled with code copied from the library, and this code would then be relocated similar to how the data from the previous section was relocated. The one exception is that it is assumed this code was already in it’s final location during the relocation calculation so ensure the code would work correctly once it was copied out of the buffer to this final location.

After the entire buffer is relocated, the relocated code is then copied from this buffer to the appropriate location in code space where it was intended to reside.

### 3.5 Library Entrypoint

After the entire library has been relocated, a predefined reference in the COFF library is used to access the new library. This reference points to a function which is used to return pointers to all of the necessary code or data of the library. This special reference is updated during the relocation process to point to it’s new location in RAM. The reference is called the entrypoint since it is where all relocated libraries begin before using the relocated library.

### 3.6 Freeing Library

When the library is no longer needed, it can be freed through a call to the Dynamic Loader. The Dynamic Loader will loop through each of the sections, de-allocating memory it had previously allocated.
4 Expectations

This run time loader will benefit in the following ways:

4.1 Abstraction Layer

Since the loader acts as another layer between the calling and the execution of the code, an API between the kernel and the libraries will need to be defined. This API is acts as an abstraction layer between the kernel and the libraries to interact with each other. The loader makes sure that despite this abstraction, the libraries are relocated correctly and pointers in the kernel are updated to reflect the library’s new location.

This will be beneficial on many levels. First, the main benefit is an abstraction of the kernel details from the libraries that are written for it. Third parties can write applications easier and more efficiently using an API instead of hard-coding references within the kernel. This helps keep the kernel safe from bugs that might be introduced while creating new libraries since no editing of the kernel is required. This will also keep prying eyes from any proprietary information in the kernel as well helping to prevent any malicious damage to the kernel since.

Secondly, changes to the kernel and the libraries can happen independently without affecting each other. As long as the API is kept constant, changes in the development of the kernel and libraries can happen in parallel without needing to know implementation details of each.
4.2 Multitasking

Allow better memory usage and allocation of data in multitasking environments. The loader, acting similar to a memory manager, adds flexibility and capability to the memory system by determining at load time where a library is placed. This ability is essential to multitasking since libraries must be able to be loaded into memory in any order or location in memory. This is because if libraries are only able to execute in a preset location, another task might be occupying a memory location while it is being loaded, preventing it’s use.

4.3 Code Reuse

Since there is an API defined for the libraries, the specific references needed by the kernel are removed and there is no need to include the kernel with each library. Only one copy is needed now, which will benefit those systems using many libraries.

4.3.1 Field Upgrades

This provides more efficient field upgrades through a simple software patch of the old library. This was more wasteful in the past since the entire library including the kernel would need to be updated even if the kernel was not changed. More seriously was if there was a change in the kernel. All the libraries must be updated since they all contain a copy of the old kernel. With a Dynamic Loader, only the kernel resident in RAM needs to be updated with the changes.
4.3.2 Code Portability

Another benefit is in the moving of libraries between different DSPs. Because of changing memory architectures, it is necessary to relink the code and kernel into a new library even if the source hasn’t changed. With a Dynamic Loader, since the libraries can be relocated into any location as long as there are sufficient resources and the memory alignment information is included in the COFF file, this extra linking step is now unnecessary. Code can be moved freely between DSPs in the form of libraries as long as they share compatible assembly languages.

4.3.3 Code Overlay

A final benefit is that not all sections of the library need be loaded at once. In order to reduce memory needs, some COFF sections need only by loaded when the system is in a mode that might make use of them. When the performance cost permits it, this could allow use of libraries that are larger than available memory by loading and running a library in smaller chunks.

4.4 Kernel Security

By removing the need to recompile the kernel for each library, third party developers do not need to have the kernel source, which results in less development time and more importantly, improved kernel security. In addition, since only one copy of the kernel is now needed, it can be placed in a small ROM within the device to prevent modification.
5 Tradeoffs

Despite the many benefits of a Dynamic Loader, there are a number of tradeoffs to adding a non-static loader to the DSP kernel. These are mainly increases in delay, complexity and memory usage. These tradeoffs can be mitigated by an efficient implementation of the loader.

5.1 Added Complexity

One obvious tradeoff is the increase of complexity and size of the kernel. The loader must be added to the kernel to provide the proper loading and relocating of the libraries. This adds even more complexity to the kernel as a level of translation must occur before code can be executed. This also adds to the kernel’s size, taking away more precious RAM from memory intensive tasks.

5.2 Delay

Another important tradeoff is the added calculations needed to initialize a library. Whereas it was merely a loading of the library, with a loader it is now necessary to load the library and then a calculate of all relocations within the library. The process, as described in Section 2.3 may involve half a dozen complex operations (i.e. multiply) for each calculation. Considering that each branch or data reference results in a relocation, this can result in a long, undesirable delay to the user while the library is being loaded.
5.3 Secondary Memory Tradeoff

The loader results in a savings in secondary memory since the kernel does not need to be repeated, but this results in a more complex library since it now takes the form of a COFF file. In the original process, the plain binaries of executable code and data was included in the library (along with the kernel). There is a savings of the kernel, but there is an added overhead associated with the COFF in the form of size, alignment, and other information. For small libraries, the size of the kernel dominates and thus there is a savings in secondary memory when dynamically loading the library (COFF). But for large libraries, the added overhead of the COFF file dominates the savings of the kernel.

5.4 Static Testability

Since memory for the library is allocated at run time, a possibility exists for this allocation to fail. This may be due to, for example, fragmentation of unallocated memory. To ensure system reliability, it is necessary to determine offline which libraries may be loaded in combination, and to guarantee this at run time. This is a topic for future work.

5.5 Further Limitations

Other minor limitations to the Dynamic Loader are described in Appendix A, Section 6. The limitations described there are mainly related to the specific implementation of the Dynamic Loader and are less to do with an inherent tradeoff of the Dynamic Loader from past library loading methods.
6 Results

6.1 MP3 Decode Library

To test the Dynamic Loader, an MP3 decode algorithm was compiled and linked into an appropriate COFF library using the method described in Section 2.2 and in detail in Appendix A, Section 3.2. The MP3 library was then loaded and relocated using the built Dynamic Loader.

Verification that the library was loaded was done by first dumping copies of the DSP memory before and after the Dynamic Loader call. The ‘before’ memory dump was verified to not contain the library and the ‘after’ memory dump was verified to contain the relocated library.

To test the memory allocation process, the memory map of the DSP was changed a number of times before the same call to the Dynamic Loader. The same before and after memory dumps were made and it was verified that the library did indeed appear in memory only after each allocation call, but the location of the library varied due to the different availability of free space before the Dynamic Load call in each example.

To verify the relocation process, a memory dump of a dynamic load was compared to a static load where sections were relocated to the same addresses. The memory dumps were equal. This result, although not a conclusive proof for all possible libraries and memory maps, showed that the Dynamic Loader achieved it’s goal by relocating all addresses in the dynamic library to the correct address references, exactly as if the library was statically loaded.
After each of these tests, the relocated library was then used to play an MP3 off of the Compact Flash card to further verify the relocation process had completed successfully.

Similar tests were performed on few other libraries, which resulted in equally successful relocations. These results proved the feasibility of dynamic loading within a DSP system.

6.1.1 Multiple Libraries

A small number of tests were done to test the loading of multiple libraries. Though time did not permit exhaustive tests, the preliminary tests indicated that two libraries were able to be relocated successfully.

6.2 Code Space

Since resources are generally limited in embedded systems, it is important to discuss the impact on these resources in addition the aforementioned benefits of a Dynamic Loader. Because of limited memory on most DSPs, code space is a large concern. Of the 128KB words available on the C5416, approximately 10KB words is used by the system code and 4 KB is used by the Dynamic Loader. Although the size of the Dynamic Loader is large in comparison to the system code, note that no attempts have been made at the present time to optimize the code space.

Optimizations aside, there is still over 85% of the RAM still available for application code and data in this test. Given that the Dynamic Loader has potentially added the ability to multitask libraries, which wasn’t possible under the static load
method, the addition of 4KB or 3% in code space appears to be a minor tradeoff for those applications that demand a multitask environment.

6.3 Load Time

Another concern for the Dynamic Loader is the added load time necessary for the relocation of the library. Now instead of a simple copy of the library from the Compact Flash to RAM, relocation calculations need to be made for each address reference within the library. The MP3 library consisted of approximately 10,000 address references. The total time to load and relocate this library was about 4 seconds compared to an almost instantaneous load in the static method.

Four seconds is clearly not an acceptable time to wait for a song to play and this is probably the largest limitation to the current implementation of the Dynamic Loader. But since optimizations were not a priority during the development of the loader, improvements can probably be made to relocation algorithm to reduce the load and relocation time of a given library. Suggestions on possible solutions this latency issue are discussed in Section 7.2. These ideas were not pursued because of the limited time frame of the project.
7 Future Work

A number of options can be explored to address the tradeoffs mentioned in Section 5 that limit the Dynamic Loader in its current implementation. Note that none of these suggestions have been implemented or tested at this stage of writing of this document.

7.1 Optimizing Code Space

For most embedded devices, RAM is very expensive and thus extra care must be taken to ensure code sizes are small enough to leave adequate space for applications, yet not sacrifice capability. The Dynamic Loader adds approximately 4KB to an already existing kernel size of 10KB. Although the current implementation of the Dynamic Loader has not been optimized for code space, a 40% increase in kernel size is still significant.

Besides aggressive optimization of code space, another interesting option is to use a similar idea of the dynamic loading on the Dynamic Loader code itself. Instead of keeping the Dynamic Loader code resident in expensive and valuable RAM where it might not be used all the time, it can be stored in less expensive ROM or secondary memory such as a Compact Flash card. When a new application is required to be dynamically loaded, the kernel could statically load the Dynamic Loader code into RAM and perform the needed Dynamic Load. When finished, the space that was occupied by the statically loaded Dynamic Loader code can be marked as data space for the application. This trick effectively frees up the 4KB of code space until a library is to be loaded.
7.2 Optimizing Load Time

The other large tradeoff of the Dynamic Loader is the increased load time for libraries. As stated earlier, the current implementation of the Dynamic Loader has not been optimized in any manner. Therefore there should be a considerable improvement in load time once optimizations are made.

7.2.1 Caching relocations

Currently the relocation algorithm of the Dynamic Loader does not take advantage of repeated addresses that require relocations. Since an address can only be relocated to a maximum of one new address, addresses that show up multiple times can be cached. The first occurrence of an address will need to be calculated to find the relocation address, but once that new address is cached, if that same original address occurs again within the library, the cached value can be used. This will result in large savings in load time for libraries with many references to the same address.

7.2.2 Caching Library

For each address reference that needs to be relocated, there is relocation data as well as symbol data that needs to be read from the library in order to calculate the new address. Since these multiple seeks and reads occur on the library which resides on slower secondary memory, benefits can be gained from caching the relocations and symbols of the library in faster RAM. This gain in speed will offset available RAM for the actual code and data of the library.
7.3 Partial Loading

One solution that addresses both code space and load time concerns is to partially load libraries. Since many libraries have distinct, predictable sections of code that are used during certain times, it will be beneficial to only load those sections that are to be used immediately and leave other sections within the library to be loaded only when they are to be used. For example, an MP3 player has initialization code which reads the MP3, sets up the bitrate and initializes variables; it also has steady state code which buffers up successive blocks of the MP3 to be decoded into an audio stream; lastly there is clean up code that frees up memory and ends cleanly. These three distinct phases of an MP3 library don’t all have to be loaded when the library is first loaded as in the current implementation of the Dynamic Loader. Loading each section only when it is needed is definitely plausible. Furthermore, because a COFF consists of separate physical “sections”, it possible to distinguish between these different parts of a library using the current COFF standard for libraries.

The benefit to partial loading of libraries is increased flexibility in how libraries are loaded. Because there is a finer control, code space issues become alleviated. Using the MP3 library example, the initialization code can be loaded initially and the code executed. When this code has finished, unload the section and load in the next section of code that is needed. This process of only partially loading the library helps to fully utilize available space in RAM, helping to alleviate the added cost of the dynamic loader code. In fact, it is theoretically possible to even use a library that is larger than available memory through partial loading.
Another benefit to the load time is the ability to break up a library into multiple loads. Although not a direct improvement in the load time to fully load a library, by breaking up the load time into multiple smaller sections may be more acceptable to the user than a large initial delay.

The ability to load separate sections within a COFF has been implemented and tested in the current version of the Dynamic Loader, although no formal performance tests were done.
8 Conclusion

Traditionally, DSP applications were only intended to be used one at a time on a specific DSP. Thus code was statically linked into a library to be used by a specific DSP platform. As long as this library was loaded into a fixed address and on the correct DSP, it would work correctly.

Today’s demands are reaching the limits of this static library method. These new applications, such as MP3 players, need the ability to multitask between multiple libraries which is difficult to do with statically linked libraries. The increasing complexities of many DSP applications also would benefit from code reuse from libraries written for other DSPs. A Dynamic Run Time Loader addresses these issues by removing the dependency of the library on a fixed address through a process called relocation. This process allows libraries to be loaded into arbitrary addresses and removes the dependence of a library on a specific DSP memory architecture.

Tradeoffs of this run-time loader include increased kernel code size and longer load times for libraries, but many options are available to address these concerns. Therefore the Dynamic Run-Time Loader for TI DSPs is an improvement in memory flexibility that will allow for improved multitasking and code reuse needed by the demanding applications of today’s embedded devices.
9 References


Appendix A
Dynamic Loader Usage Documentation

1 Introduction

1.1 Why a Dynamic Loader for DSPs
Why is a Dynamic Loader needed for DSPs? Currently in all TI DSP devices not all libraries that are needed can be stored in DSP RAM. One example is in the Internet Audio group where the libraries are the many decoders for audio: MP3, real audio, wav, windows media audio, etc. Each of these decoders is compiled and linked into a separate static library. It is important to note that by being a static library, they can only run in a fixed memory location in RAM. These libraries are stored on secondary memory (Compact Flash) until they are needed. When a library is requested, it is copied into the DSP RAM overwriting the old library and the whole DSP is rebooted to start up the new library. This process of copying statically linked libraries into the DSP works fine when dealing with only one library at a time, but when multiple libraries are needed simultaneously (i.e. an audio decoder and video decoder and graphic equalizer) and this is no longer feasible. This is because of two reasons. First, since libraries are statically linked, they can only reside in a fixed location in RAM. Two libraries can not run simultaneously if their load addresses overlap at all. Secondly, after a new library is loaded, in order to run the library, the whole DSP needs to be rebooted.

1.2 Solution
Dynamic Loader is a utility which allows libraries on secondary memory to be loaded into a dynamic location in DSP memory (RAM). In other words, the Dynamic Loader is able to load any library into any location in DSP memory provided there is sufficient space and correct alignment. The traditional static loader simply copies the needed library into a fixed location into memory. The library cannot be run if copied into a location in memory different from the linked address since all the address references within the library will be incorrect.

In contrast, the Dynamic Loader is able to copy the library into any location in memory and also fix up the address references in the library to reflect the new location. The Dynamic Loader removes many of the restrictions of statically linked library systems by adding flexibility to memory usage, especially in multitasking environments or those that can call on many different libraries.

1.3 Capabilities
This Dynamic Loader is generally able to perform the following:

- Read in COFF libraries from secondary storage
- Understand memory requirements of the sections within the library
- Allocate appropriate program and data memory for library in DSP RAM
1.4 Limitations
These are the things that the Dynamic Loader cannot do:

?? Know if your COFF library is valid
?? Know for sure if a library can fit in memory before it is loaded
?? Use FAR Mode
?? Load/Relocate only specific sections of a library
?? Play Chess

Also, please see the Notes x.1 section before attempting to use this software
2 File Structure

2.1 Files

The Dynamic Loader consists of 5 header files:
- `ia\mmdrv\src\dllobj.h` - Format of Dynamic loader algorithm object. Modeled after XDAIS standard.
- `ia\mmdrv\src\dll.h` - Implementation of Dynamic loader
- `ia\mmdrv\src\dllrelocate.h` - Relocation functions of Dynamic Loader
- `ia\mmdrv\src\dllfile.h` - File specific functions (reading and writing to secondary memory)
- `ia\mmdrv\src\dllmem.h` - Memory specific functions (allocating, reading, writing and deallocating in RAM)

(These files have a _ia suffix in the name since they were used for Internet Audio).

Support files (not specific to Dynamic Loader):
- `ialg.h` - XDAIS constants
- `coff.h` - constants and data structures for COFF format
- `ata.h` - ATA file structure (FAT 32/16) - used by `dllfile.h`
- `copy_data2prog.asm` - Assembly code used to copy data from DATA to PROG - used by `dllmem.h`
- `f.inc` - Near/Far mode calls - used by `dllmem.h`
- `rts.lib`

2.1.1 File Dependencies

Below is a rough outline of the file dependencies in the Dynamic loader.

(i.e. `dll.h` depends on `dllfile.h` which depends on `ata.h`, etc)
2.2 Classes
These 5 header files can be viewed as "classes" as in Object Oriented programming. The benefit of this class structure (besides code readability) is ease of porting to other systems because of abstraction. The Dynamic Loader Class only uses functions in the File Class to read from secondary memory, and only uses functions from the Memory Class to access any memory. Porting to another system means only changing the class that gets affected and not the whole program. For example, if one wanted to port the current Dynamic Loader to a system that uses a different file system, only the File Class (and it's dependencies) needs to change. Same for changes in the memory system.
There are three classes that make up the full Dynamic Loader:

2.2.1 Dynamic Loader Class
dllobj.h, dll.h and dllrelocate.h make up the main Dynamic Loader Class which does the loading and relocating of the code and data. The class was separated into 3 separate files to mainly simplify readability. The user should not need to edit these files.

2.2.2 File Class
dllfile.h contains the File class which is used by the Dynamic Loader Class. This File Class contains all the functions for reading from secondary memory storage (i.e. Compact Flash). The user will most likely need to edit this file for the specific file system and hardware in use. See Usage 3.x for details.

2.2.3 Memory Class
dllmem.h contains the Memory Class which is used by the Dynamic Loader Class. This Memory Class contains functions for all allocating, reading, writing and de-allocating in DSP RAM. The user will most likely need to edit this file for the specific DSP architecture used. See Usage 3.x for details.

2.3 XDAIS
The above 5 files are also structured according to the XDAIS standard. XDAIS is a standard format which DSP algorithms may conform to. Because it is a standard format it becomes easy for XDAIS algorithms written elsewhere to reused or for different algorithms to be swapped in and out of a system (i.e., testing different filters in a system).
For more information see the XDAIS specifications **********

Each of the three classes described in 2.2 above are treated as separate XDAIS objects. Each XDAIS object needs to have an interface defined as per the XDAIS specifications. For the File Class, the XDAIS interface is located in the beginning of dllfile.h. For the Memory Class the XDAIS interface is located in the beginning of dllmem.h. For the Dynamic Loader Class the XDAIS interface is located by itself in dllobj.h.

Note: Although every effort was made to conform to the XDAIS, there may be some inconsistencies from the specs. Currently there is no way to verify that an XDAIS object is valid.
3 Usage of Dynamic Loader

There are three main steps needed to get the Dynamic Loader working. Preparing Dynamic Loader, Preparing Dynamic Library, and Using Dynamic Library. They are listed here in order of decreasing complexity and are further explained below.

1. Preparing Dynamic Loader:
   1. Add Dynamic loader files into application
   2. Update DLLMem, DLLFile based on platform
   3. Customize DSP/BIOS memory constraints
   4. Update memory constraints constants in Dynamic Loader files

2. Preparing Dynamic Library
   1. Writing algorithm
   2. Compile and Link algorithm into library
   3. Add wrapper to library

3. Running the Dynamic Loader
   1. Instantiating XDAIS objects (DLLMEM, DLLFILE, DLL)
   2. Locate Library
   3. Relocating Library
   4. "Calling" Library to open wrapper
   5. Using Library
   6. Freeing Library

3.1 Preparing Dynamic Loader
The process of preparing the Dynamic Loader for use is explained here.

3.1.1 Add Dynamic loader files into application
This is pretty easy and straightforward, simply move all the Dynamic Loader files listed in 2.1 into the project that you want to have dynamic loading capability. Remember to also include the other support files that are listed at the end of 2.1.

3.1.2 Update DLLMem, DLLFile based on platform
This section will require some coding and knowledge of the DSP, file system and memory architecture the Dynamic Loader will be running on. Please see x.3, x.4 and x.5 before attempting this section.

3.1.2.1 Update DLLMEM.H
If you plan on using NEAR mode for your library, then you won't need to make many changes to this file. Essentially you will need to allocate space for the program and data
memory on the DSP through DSP/BIOS and update this in DLLMEM.H. If you need to use FAR mode, you will need to a little more work since this code has not been tested in FAR mode.

?? FAR Mode
All current memory accesses are assumed to be NEAR mode (16 bit) addresses in DLLMEM. If FAR mode capability is desired, many of the functions in DLLMEM would need to be altered to access the larger address. A __far_mode flag has already been defined in f.inc which is used by the copy_data2prog.asm code only, so this code should work in FAR mode as long as the flag is set correctly. (The copy_data2prog.asm code was adapted from the Internet Audio system code.) See Warnings 6.2 for other code changes potentially necessary when using FAR mode.

?? DYN_DATA
This parameter is the name of the data memory segment reserved in DSP/BIOS for the dynamic library. See Customizing DSP/BIOS 3.1.3 for more details on doing this. Make sure the name used in DSP/BIOS is the same as the name used in DLLMEM. The DYN_DATA parameter is used when allocating and freeing memory in the DSP RAM for DATA.

?? PROG_MEM_BASE, PROG_MEM_END
These two constants are needed to outline the program memory segment reserved for the dynamic library. Because the assembly code used to move code from data memory to program memory (copy_data2prog.asm) doesn't communicate with DSP/BIOS, it is unnecessary to "reserve" the specific segment you wish to use for program memory in DSP/BIOS. Nonetheless it is recommended to actually reserve this program memory in DSP/BIOS to keep track of your overall memory usage.

?? PROG_MEM_BASE is the starting address of the reserved program memory and PROG_MEM_END is the ending address. Set these two constants to the starting and ending addresses of the block of program memory you have allocated for the library. Currently only one contiguous section can be used at once, and deallocations of program memory must occur in the reverse order as the allocations. See details of the allocation function in the specific description of DLLMEM.H.

3.1.2.2 Update DLLFILE
This file currently supports the ATA file system (based on ata.h). If the same filesystem is chosen to be used, then no changes should be necessary to this file, skip the rest of this section. Other filesystems will need reimplementation of all the functions and part of the XDAIS interface for this file.

?? Interface
The DLLFILE_Params must be changed to the specific file handle of the new format. Replace the pointer to an ATAFile to the new file format.

?? Functions
Each of the functions in DLLFILE will need to be changed to accept the new file format:
DLLFILE_Tell_1 - replace the AtaTell call with a similar call in the new format which returns the current file pointer of a given file.
DLLFILE_Seek_1 - replace AtaSeek with a call in the new format which seeks to a new position within the file.
DLLFILE_Get_* - replace AtaRead in each of the DLLFILE_Get_* calls with a similar function that reads a word (16bits)

?? Addressing
Addresses used by the Dynamic Loader to read from files are byte addressed, but the ATA file system is word addressed. That is why in DLLFILE_Tell_1 that the file position requested was 1/2 and the opposite conversion in DLLFILE_Seek_1.
Remember to account for the addressing modes of the new format you use.

?? File Endianness
The endianness of the COFF compared to what is expected by the DSP can also be different. The COFF uses big endian (most significant byte in lowest memory address) while the DSP expects data in little endian format. Since the TI compiler outputs COFFs in big endian format, it is unlikely one will create a library using TI tools that use little endian. But if the endianness used in the COFF does change, then simply remove all of the endian flipping code near the end of the DLLFILE_Get* functions. i.e. data=((data << 8) & 0xFF00) +((data >> 8) & 0x00FF ); in DLLFILE_Get_short_1.

?? Reading strings of variable length
The only strings read in from the COFF were of all the same fixed length, so the function DLLFILE_Get_string_1 only reads in strings of length 8 characters exactly. To read in other strings, add a parameter to specify length of string to read to this function.

3.1.3 Customize DSP/BIOS memory constraints
Two memory sections, one each for program and data, should be allocated in DSP/BIOS to prepare the Dynamic Loader.

3.1.3.1 Data Memory
This section describes how to set aside memory needed by the Dynamic Loader for data in the library through DSP/BIOS.
First approximate the size of data space needed by the library to be relocated. The size information can be found in the .map file of the linked library from Compile and Link algorithm 3.2 if unsure. Be sure to only account for the data sections. If unsure about which sections are data, look at the type flag in the section header of the COFF. If the flag=0x20 (mod 0x40) then it needs program space, otherwise it resides in data space.

When approximating the data size needed, keep in mind that certain data sections will need to be aligned to specific memory addresses, so a larger memory block might need to be allocated to ensure these sections have space to be allocated. (If possible, try to move these sections to beginning of the COFF so they are allocated first).

Also include 1K of unaligned buffer space in the above data space calculation. This buffer is used by the Dynamic Loader as a temporary storage for program code. Program code is copied here for the addresses to be relocated before being copied to the permanent location in program space. (It is difficult to edit program memory). The buffer size can be changed in the DLL_H, and does not have a minimum size or alignment requirement. See Updating Memory Constraints 3.1.4 for more details.

Go into the DSP/BIOS menu in Code Composer

Select MEM – Memory Section Manager, right click and create a new section.

Name this new section the same as in the data memory parameter in DLLMEM (originally DYN_DATA).

Right click on this new MEM section and specify the approximate size needed by the library. It will be helpful to know sizes of DSP RAM as well as your system code (minus Dynamic Loader). Make sure to set the memory type to DATA.

3.1.3.2 Program Memory

While allocation for program data in DSP/BIOS is not necessary, it is recommended to do so in order to keep track of memory usage of your program.

First approximate the size of program space needed by the library to be relocated. The size information can be found in the .map file of the linked library from Compile and Link algorithm 3.2 if unsure. Be sure to only account for the program sections. If unsure about which sections are data, look at the type flag in the section header of the COFF. If the flag=0x20 (mod 0x40) then it needs program space, otherwise it resides in data space.

Go into the DSP/BIOS menu in Code Composer

Select MEM – Memory Section Manager, right click and create a new section.
Name this new section the same as in the program memory parameter in DLLMEM (originally DYN_PROG).

Right click on this new MEM section and specify the approximate size needed by the library. It will be helpful to know sizes of DSP RAM as well as your system code (minus Dynamic Loader). Make sure to set the memory type to PROG.

3.1.4 Update Memory Constraints constants in Dynamic Loader files

Two constants might need to be updated in DLL.H: MAXMEMRECS AND MAXBUFFSIZE.

MAXMEMRECS refers to the maximum number of sections that the Dynamic Loader will load. The Dynamic Loader will dynamically allocate XDAIS memRecs (from the DLL XDAIS object) for each sections in the library up to MAXMEMRECS. These memRecs are used to store the section header information pertaining to each section. If a library contains more sections than MAXMEMRECS, the entire library will not be loaded. It is provided for two reasons a) compatibility to the XDAIS specifications b) to be used by the system designer to control the size of overhead that libraries can demand, since every section needs it’s own memRec allocated for the section header information. (Each memRec takes up approximately 20 words). This constant can be changed without affecting any other aspects of the Dynamic Loader.

MAXBUFFSIZE refers to the size of the buffer reserved in data memory to be used for temporary storage for program code. Program code is copied into this buffer when a section containing program code is to be relocated. The program code address references are updated (through the relocation process) and then are copied to the permanent location in program space. This is done because program memory is difficult and inefficient to edit in place.

3.2 Preparing Dynamic Library

3.2.1 Writing Algorithm

Write the algorithm that you wish to load dynamically. This algorithm can be written the same as any other project you create in CCS. Make sure that this library executes correctly before attempting to dynamically load it.

If you already have an algorithm written or wish to import one written by a third party, make sure that it is in the correct format. See Notes x.2 to view the COFF and verify it manually. Make sure that all section headers are valid and all relocation entries for all the sections exist.
3.2.2  Library Generation
The libraries used by the Dynamic Loader must be in a specific format: COFF or Common Object File Format. COFF libraries are files which contain blocks of code and data, called sections. Besides the code or data, each section also contains information about the memory needs, alignment and relocation information for itself. This metadata information is used to fix address references when the section is relocated to a different location in memory. By relocating each of the sections that make up a library, that is how the Dynamic Loader is able to fully load a library into another location in memory. The TI assembler and linker by default creates object files in COFF (commonly with the .out extension). See Notes x.1 for more details on COFF.

The first step in using the Dynamic Loader is to create a library. A "library" is defined here as any set of code compiled/assembled and linked together. Since the TI assembler and linker in CCS (Code Composer Studio) generates output in this format, it is very simple to compile a library from any C or assembly code (See Warning 6.1) simply by building a project written in CCS. By default building a project in CCS will generate an COFF output file with the extension .out. When generating a COFF, make sure no options are set to strip away relocation entries in the build options (See Notes x.2 for help on validating the library).

3.2.3  Library Wrapper
Once generated, the library can essentially be loaded by the Dynamic Loader. But loading the library in it's current state is in a sense useless because the caller of the Dynamic Loader has no way of knowing where the requested library has been placed in memory. In order to communicate this information, we must relink the original library, say libA, with an additional function which returns a structure containing references to the original library. We can place this function at the entrance of the new library, commonly called the entrypoint of the library. This new library, say libB, will be exactly the same as libA with the addition of the small function and structure as a wrapper to the original library.

To create libB follow these steps:

?? Create a new project and include libA into the project
?? Define a structure with pointers to the needed functions or data of libA. These pointers should include all of the desired functions or data to be used as through this structure is the only way libA can be accessed after it is loaded. One suggestion is to build libA using the XDAIS specifications. Then only pointers to the handle, "MemRec" and "Params" structures of libA need to be included in the structure. Since those same structures are used by XDAIS to define an algorithm fully, they will also be sufficient to identify the library after it has been relocated. You can put this structure into a header file separate from the new project since you will need it also when calling the library after it has been relocated.

/*decode_ptr.h*/
typedef struct Decode_ptr{
    void *decode;
    void *decode_memrec;
    void *decode_params;
}Decode_ptr;

?? Declare a global instance of the above structure in this new project. Also, define a function with no parameters that returns a pointer to the above structure. This is the “wrapper function”. In this function set the pointers in the structure to the actual objects they represent. Return a pointer to the structure at the end of the function.

    Decode_obj decodeobj;

    void *decode_ptr()
    {
        decodeobj.decode=&MP3_TI_obj0;
        decodeobj.decode_memrec=&MP3_TI_obj0_memRec;
        decodeobj.decode_params=&MP3_TI_obj0_params;
        return &decodeobj;
    }

?? Set this function name to be the entrypoint to the project. This can be set in the project options from the pull down menus in CCS. Make sure you include an prefix underscore when entering the function name into the dialog box as functions are prefixed with an underscore when complied. Therefore if you defined your function as: void* libentry()… write _libentry into the dialog box for entrypoint of the project.

?? Compile and link this new library

This new library can now be relocated by the Dynamic Loader and then called which will execute the wrapper function at the entrypoint, returning the structure. Since the library was relocated, all the addresses in this structure have also been relocated to their new permanent locations, so now we know where needed code or data has been relocated in the library in the returned structure.

### 3.3 Running the Dynamic Loader

The last major step in using the Dynamic Loader is to actually instantiate and call the Dynamic Loader code with the library to be relocated. This last step should be relatively painless if you have successfully completed the previous two steps. If you have the rest of the Internet Audio code, you can look at deccfg.c for an example of how to instantiate the Dynamic Loader code and relocate the library.

In the project you are adding the Dynamic Loader to, be sure to include dll.h, coff.h and ata.h. You should also include the header file of the wrapper structure you defined in section 3.1.3.
3.3.1 Instantiating XDAIS objects (DLLMEM, DLLFILE, DLL)

3.3.1.1 DLLFILE

?? Declare the dllfile pointer, dllfile object, params structure. Declare the functions structure and initialize it with the functions in the DLLFILE.H that you defined.

```c
DLLFILE_Obj* dllfile;
DLLFILE_Obj dllfile_obj;
DLLFILE_Params params;
DLLFILE_Fxns dllfilefxns={NULL, &DLLFILE_Seek_1,
&DLLFILE_Tell_1,
&DLLFILE_Get_char_1, &DLLFILE_Get_short_1,
&DLLFILE_Get_long_1,
&DLLFILE_Get_string_1, NULL};
```

?? Next initialize the dllfile pointer with it’s associated params and functions.

```c
dllfile=&dllfile_obj;
dllfile->params=&params;
dllfile->fxns=&dllfilefxns;
```

3.3.1.2 DLLMEM

?? Declare the dllmem pointer, dllmem object, params structure. Declare the functions structure and initialize it with the functions in the DLLMEM.H that you defined.

```c
DLLMEM_Obj* dllmem;
DLLMEM_Obj dllmem_obj;
DLLMEM_Fxns dllmemfxns={&DLLMEM_Allocate_1,
&DLLMEM_Write_char_1,
&DLLMEM_Read_char_1, &DLLMEM_Write_int_1,
&DLLMEM_Read_int_1,
&DLLMEM_Write_long_1,
&DLLMEM_Read_long_1,
&DLLMEM_Write_string_1,
&DLLMEM_Read_string_1,
&DLLMEM_Free_1};
DLLMEM_Params dllmemparams={PROG_MEM_BASE};
```

?? Next initialize the dllmem pointer with it’s associated params and functions.

```c
dllmem=&dllmem_obj;
dllmem->fxns=&dllmemfxns;
dllmem->params=&dllmemparams;
```
3.3.1.3 DLL

?? Declare the dll pointer, dll object and params structure. Declare the functions
structure and initialize it with the function in DLL.H.

    DLL_Handle dll;
    DLL_Params dllparams;
    DLL_Fxns dllfxns={&DLL_Load_file_header_1,
    &DLL_Num_alloc_1,
    &DLL_Alloc_1, &DLL_Init_1, &DLL_Free_1};

?? Next initialize the dllfile and dllmem pointers in the dllparams object

    dllparams.dllfile=dllfile;
    dllparams.dllmem=dllmem;

3.3.2 Locate Library

Using the functions of your chosen file system, find the library to be relocated and assign
a pointer to it in the dllfile object. The following example uses the ATA file system (the
library to be loaded is “mp3lib.out”)

    //define atafife object
    AtaFile atafife;
    //define filename of library to be loaded
    char mp3lib[]="MP3LIB ";
    char mp3libext[]="OUT";
    //change to root directory and set the atafife to the first
    file
    AtaCdRoot(dllfile->params->atafile);
    AtaFindFirst(dllfile->params->atafile);
    //keep looping through the files until requested library is
    found
    while(1)
    {
        if
        {
            !strcmp(mp3lib,dllfile->params->atafile->Filename) &&
            !strcmp(mp3libext,dllfile->params->atafile->Ext)
        }
        {
            break;
        }
        //set atafife to the next file in the directory
        AtaFindNext(dllfile->params->atafile);
    }
    //set atafife object to pointer in dllfile object
    dllfile->params->atafile=&atafile;

Notice the last line: dllfile->params->atafile=&atafile; which actually sets the file pointer
within dllfile to the actual atafife object. Don’t forget to include this line. Error
conditions could also be included within the infinite loop in case the file is not found.
3.3.3 Relocating Library

After all of the initializing of the XDAIS objects and locating the library, you are ready to finally call the Dynamic Loader! This line calls DLL_create, passing in the functions and params of the dll object. This function will do all of the loading and relocating of the library and return a pointer to this relocated library when finished. Note: this line can take a while to execute.

dll=DLL_create(&dllfxns, &dllparams);

Congratulations, you have just dynamically relocated your library!

3.3.4 "Calling" Library to open wrapper

Remember that we had placed a wrapper around the original wrapper in order to find the functions and data after it has been relocated. So before being able to actually use the relocated library, we must call the function acting as the wrapper, which returns a structure of pointers to the library.

?? Define some temporary virtual function (f_ptr in this case).
void* (*f_ptr)();

?? Define a pointer to the structure that will be returned. The structure was named Decode_ptr and is found in decode_ptr.h.
Decode_ptr *decodeptr;

?? Set the virtual function to the entrypoint of the relocated library
f_ptr=(void * (*)(()))dll->params->entrypt;

?? Call the virtual function, which returns a pointer to the structure within the relocated library. This structure contains the correct addresses to the functions in the library.

decodeptr=f_ptr();

3.3.5 Using the Relocated Library

You can now use the relocated library through decodeptr. These three lines point to the decode object, memrec and params of the Decode XDAIS object.

decodeptr->decode
decodeptr->decode_memrec
decodeptr->decode_params

3.3.6 Freeing Library

After you have finished using your library, you will probably want to free it from memory so that you can use other libraries. Calling the following line will
DLL_Free(dll);
4 Dynamic Loading Algorithm

This section describes the process in which a library is loaded and relocated in more
detail than (Section 3.3).

It is useful to see file deccfg.c for examples on how the Dynamic Loader is initialized and
used.

4.1 Using the Dynamic Loader

4.1.1 DLLFILE Definition

First, two classes need to be defined, DLLFILE and DLLMEM.

A DLLFILE object is defined in the following four lines:

```c
DLLFILE_Obj* dllfile;
DLLFILE_Obj  dllfile_obj;
DLLFILE_Params dllfileparams;
DLLFILE_Fxns dllfilefxns={NULL, &DLLFILE_Seek_1,
&DLLFILE_Tell_1,
&DLLFILE_Get_char_1,&DLLFILE_Get_short_1,
&DLLFILE_Get_long_1,
&DLLFILE_Get_string_1, NULL};
```

The first line is merely a pointer or handle to the actual DLLFILE_Obj on the second
line, dllfile_obj. The dllfile_obj structure only contains pointers to the DLLFILE_Params
and DLLFILE_Fxns associated with this DLLFILE object.

The third line is the definition of the DLLFILE_Params, which is the structure used to
store parameters used by DLLFILE_Fxns. The main parameter located in this structure is
the filename of the library to be relocated.

The last line is the definition of DLLFILE_Fxns. If you examine the definition of the
DLLFILE_Fxns structure definition you will see that it is a list of virtual function
pointers. The assignment of the actual functions to these pointers is done on the same
line as the definition of the DLLFILE_Fxns structure. Access to the function can be done
through the virtual function name. For example, after the definition above to access the
function DLLFILE_Seek_1, one can use dllfilefxns.DLLFILE_Seek(...).

Finally the dllfile_obj is then assigned to the DLLFILE pointer (dllfile) and dllfileparams
and dllfilefxns are assigned to the params and fxns pointers respectively in dllfile_obj.

```c
dllfile=&dllfile_obj;
dllfile->params=&dllfileparams;
dllfile->fxns=&dllfilefxns;
```
4.1.2 Accessing DLLFILE
After these assignments, the entire DLLFILE object, dllfile_obj in this case, can be accessed through the pointer dllfile.

For example to access the file pointer DLLFILE object one can use:

```c
dllfile->params->atafile
```
And similarly to seek to the beginning of the file:

```c
dllfile->fxns->DLLFILE_Seek(dllfile, 0)
```

Note how the dllfile pointer is passed into the Seek function above. This is simply because it’s not possible to create a “real” class in C, so by passing in dllfile, the function is able to access the params section of the DLLFILE “class”.

4.1.3 Benefits
This method of defining “classes” is adapted straight from XDAIS. One of the benefits of this method is that the actual implementation of the DLLFILE class is hidden under the layers of pointers and virtual functions. This makes it easy to port the code if another file system is used since only the underlying parameters or functions need to be rewritten or changed, without having to change the pointers or virtual function names. Therefore any code that uses DLLFILE class won’t need to be affected when the filesystem changes, only the DLLFILE class needs to change, which makes sense.

4.1.4 DLLMEM Definition
The definition of the DLLMEM object is very similar to the DLLFILE object

4.1.5 DLL Definition
The definition of the DLL object is very similar to how DLLMEM and DLLFILE are defined except that a DLL_Obj isn’t defined and assigned to the DLL pointer as in the previous two cases. Instead, a function DLL_create is called, passing in the dllfxns and dllparams of the DLL object. This function call does the actual relocation of the library (described in Section 4.2) and then returns a pointer to the relocated DLL object. The DLL pointer, dll, is assigned to this returned value. (last line)

```c
/*DLL_OBJ*/
DLL_Handle dll;
DLL_Params dllparams;
DLL_Fxns dllfxns={&DLL_Load_file_header_1, 
   &DLL_Num_alloc_1, 
   &DLL_Alloc_1, &DLL_Init_1, };

dllparams.dllfile=dllfile;
dllparams.dllmem=dllmem;
dll=DLL_create(&dllfxns, &dllparams);
```

The reason for this difference is that in order to create the DLL Object, the number of sections within the library must be known so that enough MemRecs can be allocated. Each section is mapped to a MemRec. dllfxns is passed into the create function since
dllfxns contains functions needed to process the library. dllparams is also passed in because it contains the pointer to dllfile which is how the library containing the number of sections is accessed.

4.1.6 Finding the Library

Before the DLL Object is created, notice these lines:

```c
dllfile->params->atafile=&atafile;
AtaCdRoot(dllfile->params->atafile);
AtaFindFirst(dllfile->params->atafile);

//loop until we find the right library
while(1)
{
  if
  (!strcmp(mp3lib,dllfile->params->atafile->Filename) &&
   !strcmp(mp3libext,dllfile->params->atafile->Ext))
  {
    break;
  }
  AtaFindNext(dllfile->params->atafile);
}
```

The first line assigns the pointer to a file pointer in the DLL Object to the ATA file pointer. Notice how the subsequent lines do not access atafile directly when accessing the library. All references are through the `dllfile->params->atafile` pointer.

The second line moves the file pointer to the root directory. The third line sets the file pointer to the first file.

The next block loops through each file until the requested library is found. The file is assumed to be in the root directory.

4.1.7 Running the Library

By this point, the library has already been relocated and technically can be utilized. The problem is that now the library has relocated, it is uncertain where the pieces of the library now reside. The next few lines are used to locate pointers or “hooks” into the relocated library. The first line initializes a virtual function pointer (`f_ptr`) to the entrypoint of the relocated library. As described in 3.2, a function is placed at the entrypoint acts as the wrapper to the relocated library.

```c
f_ptr=(void (*)(()))dll->params->entrypt;
```

The last line calls the virtual function, calling the function located at the entrypoint of the relocated library. This function call simply returns a pointer to a predefined structure. This structure contains pointers or hooks which are used to access the relocated library.

```c
decodeptr=f_ptr();
```
Now the relocated library can be accessed in full through these three hooks:

```c
decodeptr->decode
decodeptr->decode_params
decodeptr->decode_memrec
```

**4.2 .cinit**

The .cinit section is a unique section within the library. It consists of special data which defines initial values for variables throughout the code and data sections of the library. For example, a normal instruction such as `int x=7;` initializes the variable `x` to `7`. Traditionally, an assembled and linked library with this instruction could be burned into a ROM, but since variables such as the integer `x` above could change, these variables had to be stored in RAM instead of unwriteable ROM. (The code could still be executed from ROM). The .cinit section stores all of these initial values of variables which are copied into RAM before the library can be used. Although the libraries in this project exist on secondary memory (Compact Flash, etc) instead of ROM, the .cinit section is still used because of adherence to the standard COFF format.

As previously mentioned, before code can be executed, the .cinit section needs to be processed. After all the code and data sections have been dynamically loaded into RAM and relocated, each initial value in the .cinit section is copied to the appropriate code or data section in RAM. Because the code and data are expected to be in RAM when the .cinit section is processed, the .cinit section must be loaded last. Also, since the .cinit section contains the section numbers and addresses of where the variable is to reside in RAM, this section must also be relocated in order to update these addresses to reflect the new locations of sections before the .cinit section is processed.
5 Code Description

This sections attempts to fully describe all the structures and functions of the Dynamic Loader, referred to as DLL in the source.

Organized by file and then just by the order the structure or function that appears in the file. There will be four main areas discussed for each function: requirements (parameters, memory, etc), functionality (what it does) and side effects (what else has changed when this function returns). Each structure’s use will be explained.

5.1 DLLOBJ_IA.H
This file contains the definition of the DLL Object.

5.1.1 DLL_MemRec
This structure is allocated for each section in the library to be loaded. This information is essentially the COFF section header information. It contains the following information about each section:

- **size**: the size of the section in words
- **alignment**: some data sections have alignment requirements
- **base location**: the final base address of the section. i.e., this is updated with the location allocated by MEM_ALLOC and will be the final location of this section (until it is unloaded)
- **v_addr**: the virtual address of the beginning of the section. In other words, the address at which the section has been static linked to. Traditionally this is where the library would be loaded. v_addr is used as a reference to calculate offsets for other address references within this section. i.e. if v_addr was 100 and another address reference, say “x”, within this section was 105, when this library was relocated to a new base address, we know that x is offset 5 words from this new base address.
- **data_file_ptr**: address of the section within the actual library file. Note that the file is in the ATA file format is also word addressed (two bytes). Despite this, the values used for seeking in the file in the source code addresses the file using byte addresses since this the format of the COFF reference documents used during development. A small speed increase might be possible by converting the few seeks within the source to word addressed values and removing the
conversions between the seek and tell functions in the DLL_FILE class.

num_reloc this is the number of relocation entries for this section
reloc_file_ptr address within the library of the beginning of the relocation entries
flag used to store the “flags” variable in the section header. This value is used during memory allocation to calculate whether a section needs to be aligned.

5.1.2 DLL_Obj
A DLL_Obj represents a Dynamic Library.

DLL_Params *params pointer to the parameters structure used by DLL_Obj.
DLL_Fxns *fxns pointer to the functions used by the DLL_Obj.
DLL_MemRec pointer to the array of memRecs for this DLL_Obj

5.1.3 DLL_Handle
This is simply a pointer to the DLL_Obj. This can be used to access the DLL_Obj.

5.1.4 DLL_Params
This structure is used to store state information regarding the library in general.

num_recs the number of total sections within this library. Note that this value also includes the actual DLL_Obj as the first section since it is allocated to be the first element of the memRec array. So if this value is 6, there are only 5 sections within the library; section 0 is the DLL_Obj. More about this in the XDAIS reference.
num_symbols total number of symbols (address references) within the library
*sym_addr physical address of first symbol entry within COFF. Symbols are located in a contiguous block near the end of a COFF. Whenever an relocation entry is looked up, a corresponding symbol for address to be relocated is also looked up within the library.
stack_sec the section number where the stack is located. Not used.
*stack physical base address of stack section in RAM. Not used.
cinit_sec  the section number of the cinit section. This section is used to store details of how variables in code or data are to be initialized.

text_sec  physical base address of first .text section found

entrypt  physical base address of entrypoint of library

*buff  physical base address of buffer used during relocation of code

DLLFILE_Obj *dllfile  pointer to DLLFILE object used for file (library) access

DLLMEM_Obj *dllmem  pointer to DLLMEM object used for RAM functions

RELOC reloc  RELOC structure used to store relocation information on current relocation

5.1.5  DLL_Fxns
Pointers to functions within DLL_Obj that are accessed by external sources. ie, these are the functions that the kernel uses to run the Dynamic Loader. Since these are just pointers to the functions defined in DLL.H, see Section XXX for the function specs.

(*DLL_Load_file_header)(DLL_Params*)
(*DLL_Num_alloc)(DLL_Params*)
(*DLL_Alloc)(DLL_Params*, DLL_MemRec*)
(*DLL_Init)(DLL_Handle)
(*DLL_Free)(DLL_Handle)
5.2 DLL.H
This file contains the actual implementation of the functions of the Dynamic Loader. The object definition of the DLL object is found in DLLOBJ_IA.H.

5.2.1 DLL_Load_file_header
DLL_Load_file_header(DLL_Params* params)

?? Requirements
takes in a pointer to the DLL_Params of the library to be loaded. The DLL_Params->dllmem->atafile pointer should be initialized with a valid COFF and the file pointer set to the beginning of the library.

?? Functionality
reads in the file header of the library, storing relevant information into DLL_Params.

?? Side Effects
file pointer DLL_Params->dllmem->atafile will be relocated to the end of the file header section.

5.2.2 DLL_Num_alloc
DLL_Num_alloc_1(DLL_Params* params)

?? Requirements
params is valid, DLL_Load_file_header has already been called

?? Functionality
returns the number of records that need to be allocated in memory. This value includes the actual DLL object itself.

?? Side Effects
none

5.2.3 DLL_Alloc
DLL_Alloc_1(DLL_Params* params, DLL_MemRec* memTab)

?? Requirements
dllparams is from a valid COFF, file headers have been loaded, memory has already been allocated for the sections headers of the COFF at memTab. Note this is different from the actual allocation of memory for the sections, which is done in DLL_Alloc_Memory
Functionality
this function initializes each of memTab with the corresponding section header information from the COFF. Size, alignment, number of relocation and flags for each section are read from the COFF and saved in the memTab of that section. References to .bss, .cinit, .stack and .text sections are also noted.

Side Effects
The file pointer will be moved to the end of the section headers

5.2.4 DLL_Alloc_Memory

DLL_Alloc_memory(DLL_Params* dllparams, DLL_MemRec* memTab, int n)

Requirements
dllparams is from a valid COFF, file headers have been loaded, n>=1, DLL_Alloc has already been called

Functionality
allocates appropriate space for each section of the COFF that is to be loaded

Side Effects
memory for at most n memRecs is allocated

5.2.5 DLL_Free

DLL_Free(DLL_Handle dll)

Requirements
dll is a valid COFF

Functionality
de-allocates all memory previously allocated by these sections

Side Effects
Frees all sections in dll from memory

5.2.6 DLL_Create

DLL_create( DLL_Fxns* dllfxns, DLL_Params* dllparams)

Requirements
dllfxns is valid, dllparams refers to valid COFF

Functionality
used to initialize and relocate a COFF library. this function is responsible for calling DLL_Load_file_header, DLL_Num_Alloc, allocation appropriate space
for the section headers for each section (memRec), calling DLL_Alloc_memory, calling DLL_Init to load and relocate the library and finally initializing the actual DLL_Obj and returning a handle to it. Note: this is the function that an external program will call when initializing the library.

??? Side Effects
returns a handle to the newly initialized library. Function fully relocates a COFF library when called.

5.2.7 Load_data
Load_data(DLL.Params* dllparams, Reloc_vars* reloc_vars)

??? Requirements
dllparams refers to valid COFF and contains valid file pointer, reloc_vars correctly initialized to contain the correct length of the data to be loaded and location of data in file.

??? Functionality
this function loads in a specific number of words from a location in file to a specific location in memory. These values are stored in reloc_vars. This function is used during the relocation phase to read in code or data from a file before performing the actual relocation.

??? Side Effects
RAM is edited with the code or data that is loaded from the file into the specified destination location in reloc_vars. file pointer in dllparams will be at the end of the section to be loaded.

5.2.8 Load_Relocate_section
Load_Relocate_section(DLL.Handle dll, int sec_num)

??? Requirements
dll refers to valid COFF, sec_num <= maximum number of sections in library, section is a loadable section

??? Functionality
loads in the given section from the library to preassigned space in memory. For data, this is the address of the memory allocated for the section from DLL_Allocate_Mem. For code this is temporarily the address of the buffer used to store code before relocation. The code or data is then relocated using the relocation information for that section. For code, the buffer is relocated assuming the code is in it’s final destination and then it is copied to the final location. This process occurs repeatedly on the buffer until the whole section has been loaded, relocated and copied to the final destination.
?? Side Effects
an entire section is loaded and relocated in RAM

5.2.9 Load_Relocate
Load_Relocate(DLL_Handle dll)

?? Requirements
dll refers to a valid COFF

?? Functionality
loops through each section in the library, calling Load_Relocate_section on those sections that are loadable

?? Side Effects
all loadable sections within a library (dll) are loaded and relocated

5.2.10 Cinit_data
Cinit_data(DLL_Handle dll)

?? Requirements
dll refers to a valid COFF, .cinit section exists for this library, library has already been loaded and relocated, including the .cinit section.

?? Functionality
processes the .cinit section specified in dll->params->cinit_sec, initializing code and data for this library

?? Side Effects
section affected by .cinit are updated in RAM with proper initial values for variables

5.2.11 DLL_Init
DLL_Init_1(DLL_Handle dll){

?? Requirements
dll refers to a valid COFF

?? Functionality
calls Load_Relocate and Cinit_data for a library. Called by DLL_Create.

?? Side Effects
same as those caused by the functions Load_Relocate and Cinit_data.
5.3 DLLRELOCATE.H
This file is essentially DLL.H, but those functions specifically related to the relocation process are located in this file for cleanliness.

5.3.1 Reloc_vars
This structure contains many of the common variables needed by Load_data (in DLL.H) and Relocate_data (in this file).

- sec_num: the number of the section that is being loaded or relocated
- data_file_ptr: address of the code or data within the file
- p_buff_base: address of where buffer is to be copied after relocation (physical address of code after relocation)
- buff_base: address of the beginning of the buffer
- length_buff: size of the buffer
- reloc_file_ptr: address of beginning of relocation information for this section within file
- length_code: total size of the code to be loaded and relocated

5.3.2 Read_Reloc
Read_reloc(RELOC* reloc, DLL_Params* params)

?? Requirements
params refers to a valid COFF, file pointer is at a valid relocation entry in file

?? Functionality
reads in one relocation entry from the current position in the file, storing it in the object pointed to by reloc

?? Side Effects
file pointer moves to end of relocation entry after reading it

5.3.3 Read_symbol
Read_symbol(SYMEMENT *sym, long sym_index, DLL_Params* params)

?? Requirements
params refers to a valid COFF, file pointer is at the beginning of symbol
entries for a section. sym_index>=0 and is valid for this section

?? Functionality
reads in the symbol at location sym_index and stores this symbol in the object
pointed to by sym

?? Side Effects
file pointer moves to the end of the symbol read in

5.3.4 relocate
relocate(DLL_Obj* dll, Reloc_vars* reloc_vars)

?? Requirements
dll is a valid COFF, current relocation entry of address to be relocated has been
read into dll->params->reloc, reloc_vars is valid and accurate

?? Functionality
this function is the heart of the relocation process. The current address reference
is relocated by this function using the relocation entry at dll->params->reloc and
reloc_vars. Essentially this function compares the base address of the section of
this address reference before and after the memory was allocated in RAM for this
section. The difference between these two addresses is the same offset for the
current address reference. For example, a branch instruction might be in a section
which starts at address 0x100. This branch instruction could be “BR 0x105” or
branch to location 0x105 within this section. When the library is first loaded, this
section could be relocated to now start at location 0x200 instead of 0x100. Since
the branch instruction is still “BR 0x105” it is incorrect since the section has now
moved to 0x200. This function will compare these section addresses and the
address reference (0x105) and rewrite the branch instruction to be “BR 0x205”,
updating the address reference.

In addition, this function must also deal with address references that point to other
sections, not just those that reference the same section.

?? Side Effects
besides the actual relocation, no other side effects should have occurred

5.3.5 Relocate_data
Relocate_data(DLL_Handle dll, Reloc_vars* reloc_vars)

?? Requirements
valid dll, reloc_vars
Functionality
this simple function loops through all of the relocation entries of the given section and calls relocate to perform the relocation process on that relocation entry.

Side Effects
file pointer is moved from reading in relocation entries, effects from calling relocate

5.4 DLLMEM.H
This file defines functions that are used to interface the main memory or RAM on the DSP such as reading from or writing to the DSP RAM. This file is separated from DLL.H for simplicity and also to add a layer of abstraction between the main code and that which interfaces with changeable parts such as RAM. Since other DSPs might have different memory structures or ways of interfacing RAM, this file hides all of those details from the rest of the Dynamic Loader code. If this code is to be ported to another DSP only this file needs to be edited to ensure compatibility with the new memory architecture.

5.4.1 Constants
PROG_MEM_BASE and PROG_MEM_END are two constants defined at the beginning of this object. They contain the permanent addresses of the beginning and end of program memory respectively. These values are the same as those allocated for program space in DSP/BIOS.

5.4.2 DLLMEM_Obj
As in the DLL_Obj structure, this structure is used as a common interface to the rest of the DLLMEM’s parameters and functions for a given DSP. Generally, only one of these objects is instantiated per DLL_Obj that is used, essentially one per DSP.

*params pointer to the DLLMEM_Params of this object

*fxns pointer to the DLLMEM_Fxns of this object

5.4.3 DLLMEM_Params
This structure contains parameters that need to maintain state within the DLLMEM object.

prog_mem_addr address of current pointer

5.4.4 DLLMEM_Fxns
This structure contains pointers to the following functions within DLLMEM:
(functions explained in further detail later)

(*DLLMEM_Allocate)(DLLMEM_Params*, int, int, unsigned int);
(*DLLMEM_Write_char)(void*, char , int );
(*DLLMEM_Read_char)(void*, int );
(*DLLMEM_Write_int)(void*, int, int );
(*DLLMEM_Read_int)(void*, int , int );
(*DLLMEM_Write_long)(void*, long, int);
(*DLLMEM_Read_long)(void*, int );
(*DLLMEM_Write_string)(void*, char*, int , int );
(*DLLMEM_Read_string)(void*, int , char*, int );
(*DLLMEM_Free)(DLLMEM_Params*, long, int, int);

5.4.5 DLLMEM_Allocate_1

DLLMEM_Allocate_1(DLLMEM_Params *dllmemparams, int size, int data_flag,
                   unsigned int align)

?? Requirements
dllmemparams, size and data_flag is valid. size and align are both small enough
to ensure allocation of memory (currently errors in allocation are not checked)

?? Functionality
given these parameters, this function will allocate the appropriate size, alignment
and type of memory in RAM

?? Side Effects
memory is allocated

5.4.6 DLLMEM_Copy_data_to_program

DLLMEM_Copy_data_to_program(long programAddress,
                           unsigned int * dataAddress,
                           unsigned int dataLengthMinusOne);

?? Requirements
programAddress, dataAddress and dataLengthMinusOne are all valid

?? Functionality
copies dataLengthMinusOne+1 words of data from dataAddress to
programAddress

?? Side Effects
program memory is changed

5.4.7 DLLMEM_Write_char_1

DLLMEM_Write_char_1(void *address, char data, int data_flag)
?? Requirements
    address, data and data_flag are valid

?? Functionality
    writes data to address in the specified type of memory

?? Side Effects
    RAM is changed

5.4.8  DLLMEM_Read_char_1

DLLMEM_Read_char_1(void *address, int data_flag )

?? Requirements
    address and data_flag are valid

?? Functionality
    reads in a char from address in the specified type of memory and returns the char

?? Side Effects
    none

5.4.9  DLLMEM_Write_int_1

DLLMEM_Write_int_1(void *address, int data, int data_flag)

?? Requirements
    address, data and data_flag are valid

?? Functionality
    writes data to address in the specified type of memory

?? Side Effects
    RAM is changed

5.4.10 DLLMEM_Read_int_1

DLLMEM_Read_int_1(void *address, int data_flag)

?? Requirements
    address and data_flag are valid

?? Functionality
    reads in an int from address in the specified type of memory and returns the int
5.4.11 DLLMEM_Write_long_1

DLLMEM_Write_long_1(void *address, long data, int data_flag)

?? Requirements
address, data and data_flag are valid

?? Functionality
writes data to address in the specified type of memory

?? Side Effects
RAM is changed

5.4.12 DLLMEM_Read_long_1

DLLMEM_Read_long_1(void *address, int data_flag)

?? Requirements
address and data_flag are valid

?? Functionality
reads in a long from address in the specified type of memory and returns the long

?? Side Effects
none

5.4.13 DLLMEM_Write_string_1

DLLMEM_Write_string_1(void *address, char *data, int size, int data_flag)

?? Requirements
address, data, size and data_flag are valid

?? Functionality
writes the string found in data with length size, to address in the specified type of memory

?? Side Effects
RAM is changed
5.4.14 DLLMEM_Read_string_1

DLLMEM_Read_string_1(void *address, int size, char * data, int data_flag)

?? Requirements
address, size, data and data_flag are valid

?? Functionality
reads in a string with length size from address in the specified type of memory
and writes this string to memory pointed to by *data

?? Side Effects
RAM is changed

5.4.15 DLLMEM_Free_1

DLLMEM_Free_1(DLLMEM_Params *dllmemparams, long address, int size, int data_flag)

?? Requirements
dllmemparams, address, size and data_flag are all valid. If freeing
program_memory (data_flag=false) then this address should be the section that
was last allocated and still is allocated. This is because all memory from this
address to the end of the program memory will be freed.

?? Functionality
this function frees the given address and size whether data or program memory.

?? Side Effects
RAM is freed. If freeing program memory, all memory from given address to end
of program memory is freed.

5.5 DLLFILE.H

This file is similar to DLLMEM.H in that it separated from DLL.H to abstract between
this code and details of the DSP system. The difference is that the abstraction now hides
details of the file system as opposed to the memory system in DLLMEM.H. Again, when
changing to a different file system (event within the same DSP) this file should be the
only class that needs to be updated. Note: no functions have been implemented that
write to the file since these features were not needed. This functionality can be added
here if needed.

5.5.1 DLLFILE_Obj

Similar to DLLMEM_Obj and DLL_Obj, this structure is used as a common interface to
the rest of the DLLFILE’s parameters and functions for a given file system used.
Generally, only one of these objects is instantiated per DLL_Obj that is used, essentially one per DSP.

*params  pointer to the DLLFILE_Params of this object

*fxns  pointer to the DLLFILE_Fxns of this object

5.5.2  DLLFILE_Params
This structure contains the parameters that need to maintain state within the DLLFILE object.

*atafile  pointer to the AtaFile object, the actual file in secondary memory. All reads and writes access this object.

str[8]  a temp string variable which is used to store strings read in from the file.

5.5.3  DLLFILE_Fxns
This structure contains pointers to the following functions within DLLMEM:
(functions explained in further detail later)

(*DLLFILE_Seek)(DLLFILE_Obj*, unsigned long);
(*DLLFILE_Tell)(DLLFILE_Obj*);
(*DLLFILE_Get_char)(DLLFILE_Obj*);
(*DLLFILE_Get_short)(DLLFILE_Obj*);
(*DLLFILE_Get_long)(DLLFILE_Obj*);
(*DLLFILE_Get_string)(DLLFILE_Obj*, char*);
(*DLLFILE_Close)(DLLFILE_Obj*);

5.5.4  DLLFILE_Seek_1
DLLFILE_Seek_1(DLLFILE_Obj *dllfileobj, unsigned long fpos)

?? Requirements
dllfileobj and fpos are valid, dllfileobj points to a valid file, fpos <=end of file

?? Functionality
this function seeks the file pointer to the address fpos within the file

?? Side Effects
file pointer in file moved

5.5.5  DLLFILE_Tell_1
DLLFILE_Tell_1(DLLFILE_Obj *dllfileobj)
?? Requirements
dllfileobj is valid and points to a valid file

?? Functionality
this function returns the address of the current file position

?? Side Effects
none

5.5.6  DLLFILE_Get_char_1
DLLFILE_Get_char_1(DLLFILE_Obj *dllfileobj)

?? Requirements
dllfileobj is valid and points to a valid file

?? Functionality
reads in a char from the current file pointer address and returns the char

?? Side Effects
file pointer in file moved

5.5.7  DLLFILE_Get_short_1
DLLFILE_Get_short_1(DLLFILE_Obj* dllfileobj)

?? Requirements
dllfileobj is valid and points to a valid file

?? Functionality
reads in a short from the current file pointer address and returns the short

?? Side Effects
file pointer in file moved

5.5.8  DLLFILE_Get_long_1
DLLFILE_Get_long_1(DLLFILE_Obj *dllfileobj)

?? Requirements
dllfileobj is valid and points to a valid file

?? Functionality
reads in a long from the current file pointer address and returns the long
?? Side Effects
    file pointer in file moved

5.5.9 **DLLFILE_Get_string_1**

DLLFILE_Get_string_1(DLLFILEObj *dllfileobj, char *dat)

?? Requirements
    dllfileobj is valid and points to a valid file

?? Functionality
    reads in a string from the current file pointer address and saves the string to the location dat

?? Side Effects
    file pointer in file moved
6 Limitations

Listed here are some of the minor limitations to the current implementation of the Dynamic Loader. Many of these issues were not addressed simply due to lack of time. Important tradeoffs and proposed solutions are mentioned in the “Texas Instruments DSP Dynamic Run-Time Loader” Section 5 and Section 7 respectively.

6.1 Error Checking

A unfortunate feature missing from this implementation of the Dynamic Loader is error-checking and proper exit status due to an unexpected error. This was due mainly to the lack time to learn the Internet Audio system’s error policies. Another reason these checks were not added was that it would significantly add to the code size of the Dynamic Loader. It was assumed that if a library had gotten into the ROM or secondary memory, it should have been verified fully before reaching a production device.

Where possible, the source has been commented to note locations where errors are likely to occur. Hopefully future developers may use these comments for debugging purposes or for further development of an error policy.

6.1.1 COFF Validation

One error check that is missing from the Dynamic Loader is the ability to verify that COFF libraries are valid. A simple verification is to check the “magic number” which is first two bytes of the library. This integer is a tag for the version of the COFF. The specific COFF used by the Dynamic Loader is the COFF2 format (generated by the C54x compiler) which has a “magic number” of 0xC2 or 192 in decimal. Checking this tag will probably be sufficient in catching the majority of invalid libraries for future developers.

Internal errors in the COFF format or malicious attempts to confuse the Dynamic Loader are also not checked. This includes everything from leaving out integral parts of the library such as the file header, discrepancies between the file header and the actual file, incorrect references to where sections begin, incorrect number of sections, relocations or size of sections, etc. Essentially, any small deviation from the COFF format, specifically the COFF2 format output by the C54x, will most likely crash the Dynamic Loader. Note that the full specifications to the COFF2 format that is generated by C54x compilers is intellectual property of Texas Instruments.

6.2 Memory Allocation

Another major limitation to the Dynamic Loader is the lack of an intelligent memory manager. Currently, it is assumed that there is sufficient RAM, in both program and data space, for the library to be loaded. Although error checking is not needed if only one library is loaded at a time, overwriting previous libraries, the lack of this feature will be a large problem once multiple libraries are loaded concurrently.
The Dynamic Loader does not analyze the library or memory to figure out if all the sections will fit in memory. Note that many sections cannot simply be placed contiguously in memory and need to be memory aligned. Additionally, the algorithm for MEM_ALLOC, the allocation instruction for the C54x could not be found, making it difficult to understand the way in which memory is allocated in the system. These factors made it hard to predict whether or not a library could be loaded, so this feature was not attempted in the Dynamic Loader.

No attempt was made to ensure optimal memory allocation of the sections of the library. Because limited knowledge is known about how future libraries are to be loaded, in what combinations and the sizes of these libraries, optimizing memory usage in this environment is extremely difficult. Since this subproblem is out of the scope of the Dynamic Loader, it is hoped that more specific research or algorithms in this field of optimal memory allocation will be used to help complete this section in the future.

6.2.1 Program Space

Another related memory allocation limitation is in program space of RAM. Due to the intricacies of the C54x DSPs, program and data space are distinct sections of the RAM. Furthermore, since code used to be simply copied into program space ala static loading, there was no need to have a memory allocation capability for program space. (There is a need for memory allocation in data space for dynamic allocations of memory for new variables and data structures). Because of this, a rudimentary memory allocation algorithm was written to allocate space for sections of a library which need to reside in program space.

This algorithm simply notes the beginning and end of a large contiguous section of program memory as available which is to be used by the Dynamic Loader for libraries. The definition of these memory sections are done in the DSP/BIOS menu. When a block of program memory is requested to be allocated, the beginning pointer to the available program memory is incremented the requested amount and that block of memory that has now been marked as used is allocated to the requested section. The next request for program memory increments the beginning pointer again. In this manner, program memory is allocated for sections in a contiguous manner. There are no memory alignment issues with program code as there is with data as long as full words are allocated at a time.

The major limitation to this method is that each allocation is not recorded, thus the Dynamic Loader only knows the locations of the beginning and end pointers to program memory. Therefore a de-allocation in the middle of the allocated program space will move the beginning pointer to address of the section, effectively freeing not only the section to be freed but also every section in between the new pointer address and the end of the program memory. In other words, sections that were allocated after this section to be de-allocated will also be de-allocated. De-allocations must happen in the reverse order they were allocated for the program memory to actually “free up”.

79
6.3 NEAR Mode
Another memory related limitation is in the amount of data memory that can be actually accessed in the current implementation of the Dynamic Loader. Because of the need for backward compatibility, two modes were created in order to address larger memory sizes. The default mode, NEAR mode is used to access up to 96K of the 128K total words in the C5416. In order to access the last 32K words, the code needs to be compiled for FAR mode access. Currently the Dynamic Loader is only able to access all of the NEAR mode memory. FAR mode addressing has not been explored. For more information on NEAR and FAR modes, see TMS320C54x DSP CPU and Peripherals – Reference Set.

6.4 Shared Data/Program memory
There are actually three sections to RAM, a program code only section, a data only section and a shared data/program section. The Dynamic Loader is able to access each of these three sections except for the extra 32K words mentioned in the section above. The actual decision as to how this shared section is split up between program or data usage (a word within this shared section can only be program or data as determined in the DSP/BIOS) must be made prior to the usage of the Dynamic Loader. These definitions must be made in the DSP/BIOS. If this memory is split in a non-contiguous way, then the DLLMEM_Allocate function must be altered to be able to allocate program and data space from more than one memory allocation. The current implementation assumes there is only one contiguous section for each program and data.

6.5 Partial Loading Limitations
As stated in the “Texas Instruments DSP Dynamic Run-Time Loader” the Dynamic Loader is capable of loading sections independent of each other. In other words, sections can be loaded one at a time from a library.

Despite this capability, there currently isn’t a way to communicate which sections are to be loaded from the library to the Dynamic Loader. There are no predefined standards to encode this information within the library and it is equally unclear how the user might be able to provide this. Therefore, the loader currently loops through each section loading and relocating it fully before moving to the next. Once the earlier concerns are addressed, it will be trivial to add an array of booleans to signify if a section is to be loaded or not to the Dynamic Loader.

Another concern that will arise if libraries are to be partial loaded and executed is the complication of certain COFF specifications. For example, a COFF library is expected to have an entrypoint defined. If multiple parts of a COFF library need the ability to be loaded separately and executed, it would be necessary to devise a way to encode all the needed entrypoints for each subset of sections loaded. Additional factors include the need for a separate .cinit and possibly a .bss section for each subset of sections to be loaded.
6.6  Fixed Program Memory Relocation Buffer

As stated in “Texas Instruments DSP Dynamic Run-Time Loader”, the relocation of the program memory uses a buffer in data memory to hold the program code as it is being relocated.

Unfortunately this buffer is currently fixed at the predefined constant of 1024 words.

#define MAXBUFFSIZE 1024

This buffer may be too large if loading a big library. The relocation should theoretically work if the buffer size was changed to a different value. Therefore it would be nice if some intelligence used to solve the memory allocation problem in Section 6.2 is also used to determine the maximum size of the buffer than can be used during the relocation of a code section. Note: Additional savings in data memory can be had by only allocating this buffer before a program section and de-allocating it right after, at a loss of speed.
7 Warnings

7.1 Arithmetic operations within address references in assembly
If the library to be relocated contains human written assembly, make sure that arithmetic operations in address references do not exist in assembly code.

ie:  BR A+B
      LD (A-B)%3

Only use static addresses such as:
    ADD C <-A, B
    BR C
    SUB D<-A,B
    LD D

7.2 FAR Mode
All code has been tested in NEAR mode. Theoretically most of this code should be similar if one chooses to use FAR mode for the DSP, but this has not been tested. Besides changes in DLLMEM, described in 3.1.2, one might also need to alter the way the COFF file is read in. This procedure is found in DLL.H and DLLRELOCATE.H. See the source for coff.exe for more details on understanding COFF. See notes x.2.
8 Notes

8.1 This document is written under the assumption the user has an adequate understanding of the COFF. See the "TMS320C54x Assembly Language Tool's - User's Guide" for more information on the COFF. It is also assumed that the user has a clear understanding of the process of relocation. Please read "TI DSP Dynamic Loader" for a full description of the process.

8.2 I have included a very rudimentary utility: coff.exe, which will help view and verify COFFs generated by Code Composer Studio. See it's documentation for more detail.

8.3 This document is written under the assumption that this software will be used in a Code Composer Studio v1.2.2 or later and the user has an adequate experience with CCS. It is also expected that the user has experience with DSP/BIOS, specifically the memory section.

8.4 The Dynamic Loader has been tested extensively on the C5416 based Leia board with Compact Flash (secondary memory) from the Internet Audio group in Dallas. Despite the testing on this specific hardware, the Dynamic Loader was designed to work with the C54x family with minimal changes to the code.

8.5 The use of the Dynamic Loader will require some changes to the code by the user depending on the system and DSP architecture used. Therefore it is expected that the user has a mature understanding of the memory architecture of the DSP, the specifications of the secondary memory system, the file format of files stored in secondary memory. It is worth repeating that it is imperative that the user understand the memory specifications of the DSP, the differences between program and data memory and how this memory is accessed. Much of the details of a DSP memory architecture can be found in the yellow user's manuals. (which one?)

8.6 If the Dynamic Loader is to be used in conjunction with other code, it is expected that the user have a rough idea of the memory uses of other system code in order to manage the memory allocation (through DSP/BIOS) and consumption between the system code and the Dynamic Loader. It is helpful to choose the linker option to output the ".map" file and analyze the memory use and mappings of the code.