Implementation of Dynamically Loaded Software Modules

Contributed by Kenneth Atwell and Joe B. *Rev 1 – May 15, 2007*

Introduction

The ability to dynamically load portions of an application from an external source at runtime can greatly increase an application's ability to respond to changing and emerging requirements within otherwise static software. This ability also allows third parties to dynamically add capability to a system without having to re-flash deployed systems.

This EE-Note introduces a framework for dynamically loading code and data at runtime. It gives consideration to file formats, run-time parsing and loading of dynamic content, run-time memory allocation, pointer relocation, debugging, and reference-by-name tactics. A documented Application Programming Interface (API) is introduced to support this framework. Example applications are provided in the file associated with this EE-Note.

The intended audience for this EE-Note is C programmers with experience using the VisualDSP++® development software. Readers should be familiar with Linker Description File (`.ldf`) fundamentals and should be comfortable with C pointers, type casting, and preprocessor macros. No specific Blackfin® processor architectural knowledge is required.

Background

Historically, a typical embedded application is represented by a single static image, created at link time and stored (e.g., in EEPROM) on the embedded device. The entire application (all of its code, data, and reserved memory for the stack and heap) is "known" at link time and is assigned memory statically by the linker.

Increasingly, it has become desirable to allow an application to dynamically link with some portion of its functionality. This may be to accommodate a non-static algorithm specification in an otherwise final product, or perhaps to allow a third party to "plug in" to the application with add-on behavior during runtime. Ideally, the memory locations used would not be hard-coded; rather, they would be assigned dynamically by the calling application. If memory locations are determined at runtime, the caller must be able to identify and make calls to these dynamically chosen locations.

Desktop operating systems like Windows® and UNIX® have permitted run-time linkage through the use of dynamically linked libraries (DLLs) and shared object (SO) files. This EE-Note suggests a light-weight framework that supports dynamically loaded modules, allowing applications to be built with less operating system support than a full dynamic-linking model would require.

Approach and Involved Files

Fundamentally, there are two sides to any application that dynamically loads a portion of its context. The main executable file (`.dxe`) represents the caller of the module. The `.dxe` is linked statically by the VisualDSP++ linker, and its contents are known entirely at link time.

The module loaded at runtime, henceforth referred to as a dynamically loaded module (DLM), represents the functionality being loaded dynamically (i.e., the "callee"). There are two requirements on a `.dxe` that wishes to load and use a DLM:

- The `.dxe` must be able to parse the DLM. The software used to perform this function is defined as the "DLM loader".
- The `.dxe` and DLM must share an agreed-upon application programming interface (API).

The bulk of this document focuses on the first requirement. Thoughtful and robust API design, the second requirement, is application-specific and is outside of the scope of this document.

In the suggested framework discussed in this document, a DLM is an executable file, but one without a C run-time (CRT) header, heap, `main()` function, and so forth. These portions are not needed as the `.dxe` provides the needed run-time infrastructure for the DLM. Although the DLM will be compiled and linked much like a "standard" executable, a DLM cannot be run as a stand-alone entity.

A `.dxe` must be able to obtain a DLM (for example, via Ethernet at runtime) and then parse and load the DLM in order to use it. A DLM can be in the same Executable and Linking Format (ELF) file format[1] as a `.dxe`, which is similar to that used by many UNIX operating systems. However, this would require embedding an ELF file reader within the `.dxe`. This is not insurmountable; in fact, many uClinux systems (among others) do this. An ELF file reader may require several hundred lines of software and may need access to a heap for the allocation of temporary data structures while parsing the ELF file. Although this approach provides maximum flexibility and compatibility, it would be costly in terms of development time, test time, and memory usage.

Instead, the suggested framework uses a very simple file format called a Binary Flat Format (BFLT) file[2], commonly referred to as a "flat file". BFLT files are quasi-standard, widely used, and have proven to be robust in uClinux systems. They are straightforward to parse (dozens of lines of code) and use memory quite efficiently, with only a small amount of overhead above and beyond the contents of the DLM itself. Analog Devices has developed a BFLT file generator, `elf2flt`[3], which is included with the VisualDSP++ tools suite, version 4.5 or later. By convention, BFLT files have a `.bflt` file extension.

The associated file contains VisualDSP++ examples that will be the basis of discussion throughout this document. Each example builds on the previous example, so examining the examples in order is recommended. Each example involves two projects collected into a VisualDSP++ group (`.dpg`) file:

- One project creates the DLM. `elf2flt` is executed as a post-link instruction in the project.

- The other project creates the "host" `.dxe`. For simplicity, and to focus the discussion of this document, the BFLT-formatted DLM is simply inserted into the `.dxe` in external memory as a "known" name. How an application actually acquires a DLM at run-time is highly application-specific (Ethernet, USB, etc.) and is not germane to this EE-Note.

These examples contain a BFLT-format parser that is used to parse and load a DLM from memory. Though the BFLT format itself and its data structures are derived from the uClinux environment, the parser in these examples is a "clean room" implementation from Analog Devices that may be used without concerns of license infringement. In order to maintain compatibility with the uClinux OS, three (and only three) section names are supported: `.text`, `.data`, and `.bss`. The Linker Description File (`.ldf`) that drives the linker during DLM creation must restrict its output to these three output sections only, which is a bit of a departure for VisualDSP++ users. The `.ldf` files included in the examples can be used as models for your own `.ldf` file.

A BFLT file begins with a small header, consisting largely of pointers into the remainder of the file. After the header is the `.text` section, followed immediately by the `.data` section. Size information for the `.bss` section is included in the header, but the `.bss` section itself is not present in the BFLT file. Since this section consists entirely of zero-initialized data, it would be wasteful of space in this data block to be actually present in the BFLT file.

A relocation table is present at the end of the BFLT file format. This critical data structure permits the DLM parser/loader to load a DLM to any location in memory, as all absolute memory pointers in the DLM are "fixed up" by the DLM loader to match the destination memory locations that are selected at runtime. This software is quite small, as there are only the three "base" addresses that the DLM loader needs to track (the locations that were selected for `.text`, `.data`, and `.bss`).

Modules with a Single Entry Point

At this time, unzip the contents of the associated file to a temporary directory and launch the VisualDSP++ IDDE (version 4.5 or later). All of the examples target the ADSP-BF537 Blackfin processor, so it is recommended that an ADSP-BF537 debug session be created (a simulator session will suffice for all of the examples related to this EE-Note) if one does not already exist. Porting the examples to other processors is straightforward because all examples use standard `.ldf` files and do not rely on processor-specific features.

Choose `File` -> `Open` -> `Project Group`, and navigate to

`…\Example_1_-_Single_Entry_Point\Example_1.dpg`.

The group file consists of two projects[1]:

- Example_1_-_DXE is the project for the `.dxe` that loads the DLM
- Example_1_-_DLM is the project for the DLM itself

Since the BFLT output of the DLM project is inserted into the `.dxe` file's image, the DLM project has been made a dependency of the DXE project.

[1] All examples associated with this EE-note follow a similar layout.

Beginning with the DLM project, it simply implements a single function to calculate the mean of an array of integers The `.dxe` loads the DLM, calls this function dynamically, and then prints (via `printf()`) the results. If you examine the `averages.c` file, you will find nothing special. It is an ordinary C implementation of a mean function.

Note the function's declaration (prototype):

```
int mean ( const int *pArray, const unsigned int dwSize );
```

This prototype will serve as the API to this function. It is important that the caller (in the `.dxe`) calls a function matching this prototype **exactly**.

More interesting is the `DLM.LDF` file, which is the `.ldf` for this DLM project. Right-click on this file in the Project window and choose `Open File.` As discussed above, only three output sections are created. Note that since the mean function utilizes an integer divide function (provided by the run-time library), the `libdsp532.dlb` library is referenced in the `.ldf` file.

Another item to note in the DLM `.ldf` file is the use of the `DYNAMIC` keyword in place of `PROCESSOR`. This feature within the linker instructs the linker to issue a `.dxe` with outstanding relocations. It is this special `.dxe` that will serve as input to the `elf2flt` tool.

Turn now to the DXE project, which gets to the heart of the BFLT framework. The bottom of the `LoadFlat.h` file contains the small API to the BFLT parser library. This API is detailed in Appendix: FLT API Reference. All functions begin with the "`FLT_`" prefix and have a return type of `FLT_RESULT`. Always check the return code for error conditions upon return (a non-zero return code indicates an error).

This example, in `main.c`, performs the following actions:

1. The size of the BFLT file is determined with the `FLT_GetSizes()` API.
2. Memory of sufficient size is reserved to hold the three sections of the BFLT file. The `.bss` section is zero-filled.
3. The new memory blocks are populated with the relocated contents of the BFLT file using the `FLT_RelocateFlat()` API.

   This example is not configured to use cache. If the relocated contents are loaded to memory areas that are cached, it is the application's responsibility to maintain cache coherency.

   The application must also be able to access the relocated contents both as data and as code. Placement of relocated contents in L1 memory is therefore illegal without the use of DMA.

4. Using a pointer returned in Step 3, a call is made to the now-relocated module. Observe that the pointer returned by `FLT_RelocateFlat()` is cast to the function pointer type that **exactly** matches the prototype of the `mean()`function implemented in the loaded DLM.

If the project group is built and run, the following `printf()` output will be displayed in the VisualDSP++ Output window (Console view):

```
Call to loaded module returned 4.
```

If further experimentation is performed, two limitations in this example become evident:

- There is no mechanism to perform C-language debugging (stepping, source breakpoints, etc.) within the DLM's source code. The debugger has full visibility into the `.dxe` file's symbol content, but knows nothing of the DLM.
- The DLM can have only a single entry point (at the top of the `.text` section).

The remainder of this document addresses these two shortcomings.

Debugging Modules

As of the 4.5 release of VisualDSP++ tools, the debugger has the ability to debug only a single executable at a time. The DLM, which is essentially its own executable, is unknown to the debugger. Programmers cannot access the following features:

- The ability to set breakpoints in source code or perform source-level stepping, both into and within the DLM
- Evaluation of local variables and global variables declared within the DLM (global labels from the "hosting" DXE are available)
- Annotation of the Disassembly window with symbol names from the DLM

We can largely address these issues by re-linking the DLM as a `.dxe` with "known" locations and then load the symbols for the DLM (now a `.dxe`). However, when the DLM's relocated symbols are loaded into the debugger, the host DXE file's symbols are lost (as the debugger only has knowledge of one set of symbols at a time). Thus, toggling between the DLM and the DXE file's symbols is possible, but working with both sets of symbols simultaneously is not.

This section details the steps needed to add debugging support to this example. If skipping manual entry of new lines of text is desired, the completed example can be found in the group file `…\Example_1_-_Single_Entry_Point_With_Debug_Support\Example_1_With_Debug_Support.dpg`.

In order to create a "static `.dxe`" version of the DLM, the application programmer must know *where the DLM sections will ultimately be loaded to on the running system*. This may be a run-time decision, so the specific locations may not be known at the project's build time. However, for debugging purposes, it may be beneficial to load the DLM to known addresses to better facilitate the strategy discussed in this document. As it turns out, the example used in the previous section of this document already has "known" destinations, the three global arrays that hold the relocated `.text`, `.data`, and `.bss` sections. Respectively, these arrays are named `g_FlatFileDestinationText[]`, `g_FlatFileDestinationData[]`, and `g_FlatFileDestinationBSS[]`. In `main.c`, the code can be implemented with a `printf()` after `FLT_RelocateFlat()`:

```
printf ( ".text at 0x%x\n.data at 0x%x\n.bss at 0x%x\nend at 0x%x\n",
    g FlatFileDestinationText,
    g FlatFileDestinationData,
    g FlatFileDestinationBSS,
    g_FlatFileDestinationBSS + dwSizeBSS );
```

However, the placement of these three arrays is likely to change between builds, as the code and data size of the project contracts and expands minutely. This fluctuation can be reduced by declaring arrays with an "absurd" alignment requirement, done using the compiler's `#pragma align` with an argument of

0x10000. The array declarations are now placed on 2^{16} boundaries, wasting upwards of 3 x 64 Kbytes of memory, but likely keeping the location of these arrays fairly constant during incremental development:

```
#pragma align 0x10000
segment ("sdram0") unsigned char g FlatFileDestinationText[ 16*1024 ];
#pragma align 0x10000
segment ("sdram0") unsigned char g FlatFileDestinationData[ 16*1024 ];
#pragma align 0x10000
segment ("sdram0") unsigned char g_FlatFileDestinationBSS [ 16*1024 ];
```

When the application is run with these two changes, the VisualDSP++ Output window (Console view) will output a message similar to:

```
.text at 0x20000
.data at 0x30000
.bss at 0x40000
end at 0x40004
```

Note the alignment at a 2^{16} boundary for all three sections (the last four hexadecimal digits are zero).

A static `.ldf` for the DLM can now be crafted, which reflects these addresses. To prevent the need to maintain two nearly identical `.ldf` files, we can use preprocessor macros to support both needs within a single `.ldf`. The `MEMORY` section of the `DLM.LDF` file is "parameterized" with a new preprocessor definition, `DEBUG_LINK` (new portions are shown in red)[2]:

```
#define DBG TEXT BEGIN   0x20000
#define DBG TEXT END     0x2FFFF
#define DBG DATA BEGIN   0x30000
#define DBG DATA END     0x3FFFF
#define DBG BSS BEGIN    0x40000
#define DBG BSS END      0x4FFFF

MEMORY
{
    // The sizes here are arbitrary and should be large enough to hold
    // the input sections they will contain.  Specific START and END are
    // unimportant as these addresses will be "abstracted" away by elf2flt.

#ifndef DEBUG LINK
    MEM TEXT { START(0x00000000) END (0x07FFFFFF) TYPE(RAM) WIDTH(8) }
    MEM DATA { START(0x08000000) END (0x0BFFFFFF) TYPE(RAM) WIDTH(8) }
    MEM BSZ  { START(0x0C000000) END (0x0FFFFFFF) TYPE(RAM) WIDTH(8) }
#else
    MEM TEXT  { START(DBG TEXT BEGIN) END (DBG TEXT END) TYPE(RAM) WIDTH(8) }
    MEM DATA  { START(DBG DATA BEGIN) END (DBG DATA END) TYPE(RAM) WIDTH(8) }
    MEM_BSZ   { START(DBG_BSS_BEGIN)  END (DBG_BSS_END)  TYPE(RAM) WIDTH(8) }
#endif
}
```

[2] This can be made smaller through the use of more command-line definitions, but it is presented this way for readability.

Since a static link will be performed, the DYNAMIC keyword must be made conditional:

```
#ifndef DEBUG LINK
DYNAMIC p0
#else
PROCESSOR p0
#endif
```

When DEBUG_LINK is defined, a static .dxe with the desired addresses will be created. When it is not defined, the DLM will be created (as has occurred up to this point in this exercise).

The final step to automate this process is to create a new post-build command to link the static .dxe whenever the DLM project is built. Right-click Example_1_-_DLM in the Project window and choose Project Options, then go to the Post-build folder. The elf2flt command line is present there. Add a second command, an adaptation of the "normal" linker command line (new options are shown in red).

```
"C:\Program Files\Analog Devices\VisualDSP 4.5\ccblkfn.exe" .\Debug\averages.doj -T
.\DLM.ldf -L .\Debug -flags-link -od,.\Debug -o .\Debug\DEBUG Example 1 - DLM.dxe -
flags-link -MDDEBUG_LINK -proc ADSP-BF537
```

After making these changes, it is now time to demonstrate our new debug capability. Build and load the .dxe project into the debugger. Open an Expressions window (View -> Debug Windows ->Expressions) and enter two expressions:

- vals
- g_iTimesCalled

vals is a global variable (an array) that is visible in the main executable, and g_iTimesCalled is a global variable *in the DLM*, which is not yet visible to the debugger. For now, this expression displays

```
ERROR: Unknown variable or symbol.
```

Now, use the File -> Load Symbols command to browse to DEBUG_Example_1_-_DLM.dxe and open this file. The g_iTimesCalled expression now evaluates without an error and vals now returns an error, evidence that the DLM's debugging information has been loaded and the callee DXE's removed. Similarly, operations like source breakpoints and C-language stepping are now permitted, but in averages.c only. Finally, the Disassembly and memory windows now contain symbols from the DLM but not the .dxe file.

Another perhaps unexpected side effect is that printf() and other debugger-based STDIO calls will not work while the DLM's symbols are loaded. This is because STDIO "works" through a silent breakpoint that is lost when the DLM's symbols are loaded. This breakpoint is subsequently restored when the .dxe file's symbols are reloaded.

The debugger can be "toggled" between the DLM and the .dxe file by loading the symbols of DEBUG_Example_1_-_DLM.dxe and Example_1_-_DXE.dxe, respectively.

Modules with Multiple Entry Points

Now that a debuggable DLM with a single entry point can be created, implementing a DLM with multiple entry points becomes of interest. Should multiple entry points be available to a caller, the caller must have a means of selecting the entry point of interest. In Windows and UNIX systems, this is accomplished by looking up the string name of the symbol (function) of interest. Lookup by name is also implemented here.

At this time, close the previous project group and choose `File` -> `Open` -> `Project Group` and navigate to …`\Example_2_-_Multiple_Entry_Point\Example_2.dpg`. Much of the detail of lookup-by-name is shielded from the caller by the `FLT` API, with only a handful of new requirements on the application developer.

First, the DLM must export a *symbol table*. A symbol table is a simple list of names (C strings) and the entry point (function pointer) associated with that string. The symbol table must be located at a known place in the application (in the case of the `FLT` API, the very top of the `.text` section). Refer to updated `DLM.LDF` file for the new input section mappings to create a symbol table at the top of the `.text` output section (with new directives shown in red):

```
.text
{
// The table of exported symbols. Each 8-byte entry in the table consists of
// a pointer to its name (string) and a pointer to its entry point (symbol)
INPUT_SECTIONS( $OBJECTS(exported_symbols) $LIBRARIES(exported_symbols))

// Terminate the exported symbol table with a NULL entry.
. += 8;

// The strings pointed to in the exported symbols section
INPUT_SECTIONS( $OBJECTS(exported_symbols_str) $LIBRARIES(exported_symbols_str))

// The program (executable code) sections itself. Exported entry points
// are pointed to in the exported symbols section

INPUT_SECTION_ALIGN(4)
INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))

} >MEM_TEXT
```

Second, the individual functions that the DLM is to export must be identified with the `EXPORT_ENTRY_POINT` macro, provided in the `export.h` header file. Here is `averages.c`, with new lines of source shown in red (a new function, `median()`, has also been provided, but is not highlighted here):

```
#include "export.h"

/* Function prototypes */

int mean ( const int *pArray, const unsigned int dwSize );
int median ( const int *pArray, const unsigned int dwSize );

/* Exported entry points */

EXPORT_ENTRY_POINT(mean);
EXPORT_ENTRY_POINT(median);
```

Note that functions exported with `EXPORT_ENTRY_POINT` must be prototyped for the compiler, which explains the ordering seen here.

With multiple entry points, the caller can no longer blindly use the top of the relocated `.text` section as a function pointer as we have been doing thus far (this would result in a jump to the symbol table itself, surely resulting in a crashed application due to execution of data as code). The caller must look up the name of the entry point of interest with the `FLT_LookupByName()` API. In the DXE project, `main.c` has been updated to use this API to access both the `mean()` function and the new function `median()`:

```
// Lookup the "mean" entry point and then call it.

if ( ret = FLT_LookupByName ( pText, "mean", (void**) &pf ) )
    return (int) ret;
result = pf ( vals, sizeof (vals) / sizeof ( int ) );
printf ( "Call to loaded module \"mean\" returned %d.\n", (int) result );

// Do the same with "median".

if ( ret = FLT_LookupByName ( pText, "median", (void**) &pf ) )
    return (int) ret;
result = pf ( vals, sizeof (vals) / sizeof ( int ) );
printf ( "Call to loaded module \"median\" returned %d.\n", (int) result );

// Now try to lookup an entry that does not exist.

ret = FLT LookupByName ( pText, "blahblahblah", (void**) &pf );
printf ( "Lookup in loaded module for \"blahblahblah\" returned %d.\n",
    (int) ret );
```

Build, load, and run the DXE project and verify that `mean()` and `median()` are called successfully, returning 4 and 2, respectively. Also, the attempt to look up `blahblahblah` failed, and `FLT_ERROR_NAMENOTFOUND (102)` was returned by `FLT_LookupByName()`.

Conclusion

Applications are increasingly requiring the ability to dynamically load some portion of their content, including executable code. There are several reasons for this, including third-party software integration, adaptation to evolving standards, and enhancement of already-deployed systems. Flat files and the FLT API discussed in this document provide a simple mechanism to load dynamic content to a running system. The software needed to parse and load FLT files is simple and compact, making it a compelling solution for embedded systems. Entry points into dynamically loaded content are indexed by name, allowing for easy look-up. Function calls to dynamically loaded code are performed through C function pointer calls and are only marginally less efficient than a “hard-linked”, static call. The VisualDSP++ 4.5 tools include a FLT file creation tool.

This document introduced debugging techniques, allowing debugging of a module that was loaded “outside” the debugger’s normal awareness.

Appendix: FLT API Reference

FLT_GetSizes

```
FLT_RESULT FLT_GetSizes        (const FLT_FILE pFlat, unsigned long *pdwSizeText,
                               unsigned long *pdwSizeData, unsigned long *pdwSizeBSS)
```

Arguments

Name	In/Out	Description
pFlat	[in]	A pointer to a memory buffer containing the complete image of a BFLT file.
pdwSizeText	[out]	A pointer to receive the size of the .text section of the BFLT file.
pdwSizeData	[out]	A pointer to receive the size of the .data section of the BFLT file.
pdwSizeBSS	[out]	A pointer to receive the size of the .bss section of the BFLT file.

Return Value

`FLT_ERROR_NOERROR` (0) on success.

Non-zero on an error.

Discussion

Examines the header of the BFLT file pointed to by `pFlat` and returns the lengths of the `.text`, `.data`, and `.bss` sections of the BFLT file.

Space for the contents of the `.bss` section is not contained in the BFLT file's image. Should the value returned in `pdwSizeBSS` be non-zero, the caller should take steps to allocate memory and zero-fill this memory, even if the BFLT file will otherwise be relocated "in place" with the `FLT_RelocateFlatInPlace` API.

FLT_LookupByName

```
FLT_RESULT FLT_LookupByName   (const FLT_SECTION pText, const char *pName,
                               void **ppSymbol)
```

Arguments

Name	In/Out	Description
pText	[in]	A pointer to the top of the .text section, where the exported symbol table should be located.
pName	[in]	A pointer to a NULL-terminated string indicating the symbol to look up.
ppSymbol	[out]	A pointer to a void pointer in which the looked up symbol is returned. The nature of this pointer is implementation-specific.

Return Value

`FLT_ERROR_NOERROR` (0) on success. `*ppSymbols` contains a pointer to the symbol found.

`FLT_ERROR_NAMENOTFOUND` on failure. `pName` was not found in the symbol table.

Other non-zero on some other error.

Discussion

Performs a search through the exported symbol table for the `pName` symbol. If found, a pointer to the symbol's location is written to `*pSymbol` and `FLT_ERROR_NOERROR` is returned. If not found, `FLT_ERROR_NAMENOTFOUND` is returned.

`pText` is a pointer to the top of the `.text` section, where the exported symbol table is found.

Symbols are exported in a DLM through the use of the `EXPORT_ENTRY_POINT` macro. Furthermore, the input section `exported_symbols` must be placed topmost into the `.text` output section in the DLM's `.ldf` file. The input section `exported_symbols_str` must also be mapped in the DLM's `.ldf` file (at an arbitrary location).

FLT_RelocateFlat

```
FLT_RESULT FLT_RelocateFlat   (const FLT_FILE pInFlat, FLT_SECTION pText,
                               FLT_SECTION pData, const FLT_SECTION pBSS,
                               void **ppText)
```

Arguments

Name	In/Out	Description
pFlat	[in]	A pointer to a memory buffer containing the complete image of a BFLT file
pInText	[in/out]	A pointer to the memory buffer to be populated with the relocated contents of the `.text` section.
pInData	[in/out]	A pointer to the memory buffer to be populated with the relocated contents of the `.data` section.
pInBSS	[in]	A pointer to the memory buffer containing the `.bss` section.
ppText	[out]	A pointer to a pointer that receives the address of the top of the `.text` section of the BFLT file. For modules with a single entry point, this is a pointer to that entry point. For modules with multiple entry points (and thus a look-up table), this is a pointer to that look-up table for a subsequent call to `FLT_LookupByName`.

Return Value

`FLT_ERROR_NOERROR` (0) on success.

Non-zero on an error.

Discussion

Copies and performs relocations on a BFLT file. `pFlat` must point to a complete BFLT file image. `pText` and `pData` are pointers to memory buffers to be populated with the relocated contents of the `.text` and `.data` sections respectively. `pBSS` is a pointer to zero-initialized memory for the `.bss` section; and relocations are performed *against* this memory, but the buffer itself is unmodified by this function. It is the caller's responsibility to zero-init the buffer pointed to by `pBSS`.

Allocation of these three buffers is the caller's responsibility. The required sizes of these three buffers can be determined with the `FLT_GetSizes` API. If any of the buffers reside in cacheable area of memory, it is the application's responsibility to maintain cache coherency after a call to `FLT_RelocateFlat`.

As the contents pointed to by `pFlat` will be copied elsewhere before any operation is performed, it is safe to place the contents pointed to by `pFlat` into non-volatile memory.

FLT_RelocateFlatInPlace

```
FLT_RESULT FLT_RelocateFlatInPlace  (FLT_FILE pInFlat, const FLT_SECTION pBSS,
                                     void **ppText)
```

Arguments

Name	In/Out	Description
pFlat	[in/out]	A pointer to a memory buffer containing the complete image of a BFLT file
pBSS	[in]	A pointer to the memory buffer where `.bss` (zero-fill) data will reside. If `pFlat` points to a BFLT file that does not contain `.bss` data, this parameter may be `NULL`.
ppText	[out]	A pointer to a pointer that receives the address of the top of the `.text` section of the BFLT file. For modules with a single entry point, this is a pointer to that entry point. For modules with multiple entry points, this is a pointer to the symbol table for subsequent calls to `FLT_LookupByName`.

Return Value

`FLT_ERROR_NOERROR` (0) on success.

Non-zero on an error.

Discussion

Performs relocations on a BFLT file "in place" on the memory buffer pointed to by `pFlat`. `pFlat` must point to a complete BFLT file image. Note that space for `.bss` sections is not reserved in a BFLT file, so any call to this function where the BFLT file has a non-zero sized `.bss` section must provide an appropriately-sized buffer that is pointed to by `pBSS`.

References

[1] The Executable and Linking Format (ELF), *http://www.cs.ucdavis.edu/~haungs/paper/node10.html*

[2] uClinux - BFLT Binary Flat Format, *http://www.beyondlogic.org/uClinux/bflt.htm*

[3] *elf2flt usage, "`elf2flt -help`" on the command line*

Document History

Revision	Description
Rev 1 – May 15, 2007 by K. Atwell and Joe B.	Initial Revision.