アナログ・デバイセズの DSP、プロセッサ、開発ツール用テクニカル・ノート
http://www.analog.com/jp/ee-notes 、http://www.analog.com/jp/processors にはさまざまな情報を掲載しています。

第 3 世代 SHARC®ファミリー・プロセッサでのコード・オーバーレイ

著者: Brian M. and Divya S.

Rev 2—2005 年 4 月 11 日

はじめに

オーバーレイは、VisualDSP++®開発ツール内で DSP 内部メモリを効率的に管理する環境を提供します。これまでリリースされた SHARC®ファミリー・プロセッサと同様に、最新の第 3 世代 SHARC プロセッサでもコード・オーバーレイを使って比較的小さい内部メモリ・ブロック(合計 1〜3 M ビット)を有効利用することができます。

これまでのオーバーレイについて記述した EE ノートを拡張したこの資料では、第 3 世代 SHARC DSP をサポートするために必要なオーバーレイについて説明します。先に進む前に、まず、次のドキュメントを読んでオーバーレイの概念を理解してください。

- *Using Memory Overlays (EE-66)* [1]
  このドキュメントでは、SHARC プロセッサ上でのオーバーレイの使用方法を説明しています。外部 RAM からのコード・オーバーレイ使用の概念と詳細が記載されています。
- *Using Code Overlays from ROM on the ADSP-21161N EZ-KIT Lite (EE-180)* [2]
  このドキュメントでは、EE-66 で説明した概念を拡張して、安価な ROM 製品からのコード・オーバーレイの使用方法が説明されています。

第 3 世代 SHARC プロセッサ(ADSP-2126x/2136x)の場合、I/O プロセッサは 48 ビット・アドレスをアクセスしなくなりました。これは、オーバーレイにおいても 32 ビット・アドレッシングを前提として使用する必要があることを意味します。48 ビット・ワードの開始または終わりが 32 ビット・ワード内に位置するため、転送を正しく行うためには特別なケアが必要となります。

このドキュメントでは、この新しい動作に対応するため、次の 3 つのプロジェクトを使ってオーバーレイ・マネージャをデモストレーションします。

- オーバーレイを格納する外部 SRAM を使ったシンプルなオーバーレイ・マネージャ
- パラレル EEPROM またはフラッシュにオーバーレイを格納するプロジェクト
- SPI フラッシュにオーバーレイを格納するプロジェクト

オーバーレイの簡単な説明

VisualDSP++が提供するオーバーレイ機能を使う場合、リンカーはオーバーレイ・セクション内に配置された関数に対する呼び出しをオーバーレイ・マネージャに対する呼び出しへと変更します。次に、オーバーレイ・マネージャは、所望のコードを外部メモリ・ロケーション(ライブ・アドレス)からオーバーレイ用に設けた内部メモリ領域(実行アドレス)へ転送し、該当するロケーションへジャンプしてコードを実行します(SHARC プロセッサ上のオーバーレイの説明については EE-66 を参照してください)。

オーバーレイが外部 RAM に格納されている場合、オーバーレイ・ライブ空間はブート時に内部メモリと同じ方法で初期化されます(ブート・ローダ・カーネルによる初期化)が、RAM の代わりに ROM またはフラッシュ・デバイスを使用する場合、ローダを使ってメモリへ書き込むブート・イメージを生成する方が容易です(EE-180 参照)。この方法を使う場合、初期化ルーチンを実行してオーバーレイが存在するロケーションと、オーバーレイ全体を構成するローダ・セクション数を調べます。

このドキュメントでは、これまで議論してきた例を元に，第 3 世代 SHARC プロセッサにおいていかにそれらを拡張するか、その方法について説明します。

オーバーレイの基本例

本 EE ノートでは、EZ-KIT Lite 上で動作する添付のオーバーレイプロジェクトに沿って説明されます。各オーバーレイは、直前に実行されたオーバーレイを表示するボード上の 8 個の LED の独自なサブセットを設定する関数を持っています。

外部 RAM からのオーバーレイ

まず、第 3 世代 SHARC プロセッサ向けに VisualDSP++で提供される外部メモリのサポートは、EE-180 と EE-66 で説明した ADSP-21161 と ADSP-21065L 向けに提供されたものとは異なっています。オーバーレイを外部メモリへ配置するときは、PACKING()コマンドを使ってオーバーレイ・データを外部メモリへ配置します。図 1 と図 2 に、それぞれ 8 ビットと 16 ビットの外部 SRAM モジュールに対して必要な PACKING()コマンドを示します。

```
PACKING(6 B0 B0 B0 B6 B0 B0 \
          B0 B0 B0 B5 B0 B0 \
          B0 B0 B0 B4 B0 B0 \
          B0 B0 B0 B3 B0 B0 \
          B0 B0 B0 B2 B0 B0 \
          B0 B0 B0 B1 B0 B0)
```

図 1. 8 ビット・メモリ用の PACKING()コマンド

```
PACKING(6 B0 B0 B5 B6 B0 B0 \
          B0 B0 B3 B4 B0 B0 \
          B0 B0 B1 B2 B0 B0 )
```

図 2. 16 ビット・メモリ用の PACKING()コマンド

これらの PACKING()コマンドは LDF ファイルの OVERLAY_INPUT()セクション内にあって、実行可能ファイル内でのデータの配置方法をリンカーに知らせる必要があります(ADSP-2126x ファミリー DSP の外部メモリについては、「*Using External Memory with Third Generation SHARC Processors and the Parallel Port (EE-220)* [3]」を参照してください)。

前の SHARC DSP からのもう 1 つの変更は、第 3 世代 SHARC プロセッサでは 48 ビット DMA 転送がなくなっていることです。この変更は、オーバーレイ・マネージャに大きな影響を与えます。パラレル・ポートまたは SPI を使うすべての転送が 32 ビット・ワードに限定されるため、48 ビットの実行アドレスを等価な 32 ビット・アドレスへ変換する必要があります。これは、48 ビット・アドレスに 1.5 を乗算することにより行われます。図 3 に、48 ビット・アドレスを等価な 32 ビットへ変換するアセンブリ・コードを示します。

```
r3=0x80100;   /* 48-bit address */
r2=3;    /* Use R2 as a multiplier */

/* Extract lower bits from R4 into R3
(bits that need to be translated) */
r3=fext r4 by 0:16;

/* Place remaining bits in R0 (these
bits indicate word space) */
r0=r4-r3;

/* Multiply lower bits by 3 */
r3=r3*r2 (ssi);

/* Divide the result by 2 */
r3=lshift r3 by -1;

/* Insert the new lower bits */
r0=r0 or fdep r3 by 0:16;
/* Finished! R0=0x80180 */
```

図 3.オーバーレイ・マネージャで必要とされる 48 ビット・アドレスから 32 ビット・アドレスへの変換

また、48 ビット・ワード数(実行ワード・サイズ)を等価な 32 ビット・ワード数へ変換することも必要です。DMA トランザクションを正常に完了させるためには、全 32 ビット・ワード数を転送するように DMA を設定する必要があります。このため、48 ビット長を 32 ビット長へ変換する場合、計算で丸め処理が必要です。

各 48 ビット・ワードは、等価な 32 ビット・アドレスの中間で始まるか終わるような形態となります。48 ビット実行アドレスが奇数の場合は、等価な 32 ビット・アドレスの先頭の 16 ビットを待避させた後にオーバーレイをフェッチして、転送が完了した後に戻す必要があります。これは、32 ビット・ワードの後ろの半分が実行アドレスに必要であるためです。同様に、実行空間が偶数の 48 ビット・アドレスで終わる場合、等価な 32 ビット・アドレスの後ろの 16 ビットを待避させ、同じ方法で戻す必要があります。このドキュメントに示す例では、開始と終わりの実行アドレスが各オーバーレイについてチェックされ、問題のアドレスが 1 ロケーションのスタックに待避され、上述のように戻されています。

実行アドレスが奇数の場合は、外部メモリが 8 ビットであるか 16 ビットかに応じて、開始ライブ・アドレスをそれぞれ 1 または 2 ロケーションだけ小さくする必要があります。これにより、48 ビットの実行アドレスとライブ・アドレスのロケーションが揃います。これは、アクセス対象の 32 ビット実行アドレスは、必ず 32 ビット境界上に存在する必要があるためです。

第 3 世代 SHARC DSP は 8 ビットと 16 ビットの外部メモリをサポートしているため、提供されているオーバーレイ・マネージャが使用する外部メモリの幅を自動的に調べます。これは、実行ワード・サイズとライブ・ワード・サイズを比較することにより行われます。

ROM/フラッシュからのオーバーレイ

ROM/フラッシュからのオーバーレイの使用は、RAM からの場合と良く似ています。オーバーレイが外部メモリに格納されているかのように、LDF ファイルの `OVERLAY_INPUT()` セクションを設定しますが、オーバーレイは、物理メモリ・デバイスによりコピーされない外部メモリ・マップ (`0x2000000`～`0x2FFFFFF`)内のダミー・ライブ・アドレスに配置します(実際のライブ・アドレスは ROM 内のロケーションにあることに注意してください。このダミー・ライブ・アドレスは ROM 内の各オーバーレイを探す際に検索文字列としてのみ使用されます。この情報はリンク時には存在しないためです)。メモリが 8 ビットであるかまたは 16 ビットであるかは問題になりません。これは、ローダがいずれのメモリ・タイプからも正しいフォーマットを生成するためです。

ローダにより生成されたイメージを使う場合、データ・セクションの前にタグが挿入されて、ブート・ストリーム内の各セクションを識別します。ダミー・ライブ・アドレスはデータ・タイプ・タグやワード・カウントと一緒にブート ROM 内に存在するため、各オーバーレイの前に配置されます。オーバーレイ・セクションを呼び出す前に、オーバーレイ・マネージャはブート・ストリームを調べて、これらのダミー・ライブ・アドレスを見つけて、ROM 内のロケーションとオーバーレイ・セクションが占有している全アドレス範囲のブート・セクション数を待避させる必要があります。オーバーレイ・セクションは、LDF ファイルで指定されるオーバーレイ範囲内のアドレスにある通常の外部メモリ・タイプを使って、タグ付けされます(表 1 参照)。

Memory Type	Tag for Zero Init	Tag for Regular
8-bit	0x7	0x9
16-bit	0x8	0xA

表 1. ADSP-2126x DSP に対するローダ・タグ

EE-180 で説明するように、オーバーレイ初期化サブルーチンがブート・ストリームを調べてオーバーレイ・セクションのロケーションを探し、オーバーレイ・セクション情報サブルーチンが各オーバーレイ・セクションを構成するローダ・セクション数を調べます。添付コードでは、オーバーレイが DSP のブートに使われる同じデバイス内に格納されていると想定しています。このため、オーバーレイ初期化ルーチンは、カーネルの後ろの最初のバイト(ロケーション `0x600`)からローダ・ファイルの解釈を開始します。

これらの例に添付されているオーバーレイ初期化ルーチンでは、すべてのオーバーレイのライブ空間は連続であると仮定しています。このルーチンは、先頭のオーバーレイのライブ開始アドレスと最後のオーバーレイのライブ最終アドレスとの間に存在するタグを探します。このルーチンはリンカーから取得したアドレスを自動的に使いますが、使用するオーバーレイ数に合わせて変更する必要があります。

オーバーレイ・マネージャを使う場合は、オーバーレイ初期化ルーチンとオーバーレイ・セクション情報ルーチンが、全 32 ビット・ワード数と有効ライブ・アドレスを自動的に提供します。ROM から実行する場合には、48 ビット奇数/偶数問題のチェックに使用した同じ計算を行って上記の実行アドレスを求める必要がありますが、オーバーレイが奇数アドレスから開始される場合は、フェッチ対象オーバーレイ内の各ローダ・セクションの先頭が 32 ビット・ワードの中間に位置するため、各フェッチ対象ローダ・セクションに対してワードの先頭 16 ビットを待避/回復させる方式を適用する必要があります。

最後に、ローダ・カーネルの場合と同様に、初期化セクションがゼロであることが検出されたとき、ゼロを転送するのではなく、オーバーレイ・マネージャが正しいワード数を自動的に初期化します。

パラレル ROM/フラッシュ・オーバーレイ・マネージャ

パラレル ROM/フラッシュ向けに提供されるオーバーレイ・マネージャは、パラレル・ポート DMA レジスタを使って内部実行アドレスを格納しますが、IIPP レジスタに待避されるアドレスは、アドレスの最上位ビットがレジスタに格納されないようにするため、32 ビットの通常のワード空間内にある必要があります。待避/回復する開始アドレスはこのレジスタから取得されるため、上位ビットをマスク・インする必要があります。この例では、すべてのオーバーレイはブロック 0 (アドレス `0x80000`～`0x9FFFF`)で実行されるものと想定しています。すなわち、マスクを常に `0x80000` とします。

オーバーレイ・マネージャとオーバーレイ初期化ルーチンで使用される `read_data_bytes()` ルーチンでは、使用するデバイスはバイト幅であると仮定しています。これは、このフォーマットを DSP のブートで使用する必要があるためです。

SPI ROM/フラッシュ・オーバーレイ・マネージャ

SPI フラッシュにオーバーレイを格納する基本的な考え方は、前述のパラレル・フラッシュに格納する場合と同じですが、SPI フラッシュをアドレス指定する方法をサポートするために、簡単な変更が必要です。

「*Programming an ST M25P80 SPI Flash with ADSP-21262 SHARC DSPs (EE-231)* [4]」で説明するように、SPI フラッシュはブート可能とするために LSB ファースト(LSBF)フォーマットで書き込む必要がありますが、SPI フラッシュへ送られるコマンドは MSB ファースト・フォーマットである必要があります。このため、SPI フラッシュに格納されたオーバーレイをアクセスするときは、LSBF フォーマットで格納されているオーバーレイ・データを読み出すためにフラッシュへ送信される読み出しコマンドとアドレスをビット反転してください。SPI フラッシュのアクセスに使用される `read_data_bytes()` ルーチンは、EE-231 で提供された SPI フラッシュプログラマから取り出したものです。

SPI フラッシュでは余分に 32 ビット転送が必要なため(ダイレクト・アドレス・ラインではなく)、SPI フラッシュの各アクセスに対して、実行空間へさらに 32 ビット・ワードを 1 ワード余分に(1 バイトの読み出しコマンドと 24 ビット・アドレスに対応して)転送します。オーバーレイ・セクションを転送するとき、このワードは書き込み対象セクションの直前にあるワードを上書きします。この影響を受けるワードは、奇数 48 ビット境界にあるワードと同じ方法で、待避/回復されます。

結論

コード・オーバーレイの概念は EE-66 と EE-180 で説明されていますが、このドキュメントではこれらの概念を第 3 世代 SHARC プロセッサまで拡張します。第 3 世代 SHARC プロセッサ・システムで 48 ビット DMA サポートがないことによるオーバーレイ使用での主な相違は、これにより容易に解決できます。

外部 RAM および ROM またはパラレル・ポートを介したフラッシュ・デバイスからのオーバーレイの使用(SHARC ファミリーの前のメンバーに対して説明したように)の他に、DSP オーバーレイは SPI フラッシュ・デバイスにも格納することができます。このドキュメントは、第 3 世代 SHARC DSP をサポートしている SPI フラッシュからのオーバーレイを使用する枠組みを提供します。この同じ枠組みは、ADSP-21161 をサポートするように拡張することもできます。

アペンディックス

ovly_mgr.asm

```
/*  The OVLY_MGR.ASM file is the overlay manager.  When a symbol   */
/*  residing in overlay is referenced, the overlay manager loads   */
/*  the overlay code and begins execution.  (This overlay manager  */
/*  does not check to see if the overlay is already in internal    */
/*  memory.)  A DMA transfer is performed to load in the memory    */
/*  overlay.                                                       */

#include <def21262.h>

.SECTION/DM             dm_data;

/*  The following constants are defined by the linker.   */
/*  These constants contain the word size, live location */
/*  and run location of the overlay functions.           */

.EXTERN _ov_word_size_run_1;
.EXTERN _ov_word_size_run_2;
.EXTERN _ov_word_size_run_3;
.EXTERN _ov_word_size_run_4;
.EXTERN _ov_word_size_live_1;
.EXTERN _ov_word_size_live_2;
.EXTERN _ov_word_size_live_3;
.EXTERN _ov_word_size_live_4;
.EXTERN _ov_startaddress_1;
.EXTERN _ov_startaddress_2;
.EXTERN _ov_startaddress_3;
.EXTERN _ov_startaddress_4;
.EXTERN _ov_runtimestartaddress_1;
.EXTERN _ov_runtimestartaddress_2;
.EXTERN _ov_runtimestartaddress_3;
.EXTERN _ov_runtimestartaddress_4;

/*  Placing the linker constants in a structure so the overlay  */
/*  manager can use the appropriate constant based on the    */
/*  overlay id.                          */
#define PHYS_WORD_SIZE(run_size,live_size) (48 / (live_size/run_size))
.import "OverlayStruct.h";

.struct OverlayConstantsList _OverlayConstants = {
_ov_startaddress_1, _ov_startaddress_2,_ov_startaddress_3, _ov_startaddress_4,
_ov_runtimestartaddress_1, _ov_runtimestartaddress_2,_ov_runtimestartaddress_3,
_ov_runtimestartaddress_4,
_ov_word_size_run_1,_ov_word_size_run_2,_ov_word_size_run_3,_ov_word_size_run_4,
_ov_word_size_live_1,_ov_word_size_live_2,_ov_word_size_live_3,
_ov_word_size_live_4,
PHYS_WORD_SIZE(_ov_word_size_run_1,_ov_word_size_live_1),
PHYS_WORD_SIZE(_ov_word_size_run_2,_ov_word_size_live_2),
PHYS_WORD_SIZE(_ov_word_size_run_3,_ov_word_size_live_3),
PHYS_WORD_SIZE(_ov_word_size_run_4,_ov_word_size_live_4),
};

/*  software stack to temporarily store registers corrupted by overlay manager */
.VAR  ov_stack[20];
.VAR  start_addr_stack [2] = 0,0;
.VAR  end_addr_stack [2] = 0,0;

/*****************************************************************/
/*          Overlay Manager Function                             */
```

```
.SECTION/PM            pm_code;

_OverlayManager:
.GLOBAL _OverlayManager;

/*  _overlayID has been defined as R0.  R0 is set in the PLIT of LDF.   */
/*  Set up DMA transfer to internal memory through the external port.   */

/*  Store values of registers used by the overlay manager in to the     */
/*  software stack.                                                     */
dm(ov_stack)=i7;
dm(ov_stack+1)=m7;
dm(ov_stack+2)=l7;
i7=ov_stack+3;
m7=1;
l7=0;

dm(i7,m7)=i8;
dm(i7,m7)=m8;
dm(i7,m7)=l8;
dm(i7,m7)=r2;
dm(i7,m7)=r3;
dm(i7,m7)=r4;
dm(i7,m7)=r5;
dm(i7,m7)=r6;
dm(i7,m7)=ustat1;

/* Use the overlay id as an index (must subtract one)  */
R0=R0-1; /* Overlay ID -1 */
m8=R0;   /*  Offset into the arrays containing linker  */
         /*  defined overlay constants.                */

r0=i6;  dm(i7,m7)=r0;
r0=i0;  dm(i7,m7)=r0;
r0=m0;  dm(i7,m7)=r0;
r0=l0;  dm(0,i7)=r0;

l8=0; // Clear L0 & L8 to keep DAGs from using Circ Buffers
l0=0;
r2=3; // Save multiplier to convert 48-bit address to 32-bit
r6=1;

m0=m8; /* Overlay ID - 1 */

/*  Get overlay run and live addresses from memory and use to */
/*  set up the master mode DMA.                               */
i8 = _OverlayConstants->runAddresses;
i0 = _OverlayConstants->liveAddresses;

r0=0;           dm(PPCTL) = r0; dm(start_addr_stack)=r0; dm(end_addr_stack)=r0;
r0=dm(m0,i0);   dm(EIPP)=r0;

//Convert 48-bit run addr to 32-bit addr (48 * 3/2 = 32)
r4=pm(m8,i8);

r3=fext r4 by 0:16; r0=r4-r3;
r3=r3*r2 (ssi);
r3=lshift r3 by -1;
r0=r0 or fdep r3 by 0:16;
dm(IIPP)=r0;

/* If 48-bit address is odd, save first 16-bits to restore after the
   overlay is loaded.*/
btst r4 by 0; if not sz jump addr_end;
r0=lshift r0 by 1;  dm(start_addr_stack)=r0;   // Store short word address
```

```
i6=r0; r0=dm(0,i6); dm(start_addr_stack+1)=r0; // Store the affected word
r6=lshift r6 by 1;

addr_end:
i0=_OverlayConstants->runNumWords;  /* Number of words stored in internal memory */
                /*  Most likely the word size will be 48 bits  */
                /*  for instructions.              */

i8=_OverlayConstants->liveNumWords; /* Number of words stored in external memory */

r0=pm(m8,i8);
i8=_OverlayConstants->physicalLen;
ustat1=pm(m8,i8);
/* Make sure that there is a whole number of 32-bit words in ECPP
   (and round up if there is not)*/
r0=r0+1; r0=lshift r0 by -1; bit tst ustat1 16; if tf jump (pc,4);
r0=r0+1; r0=lshift r0 by -1; r0=lshift r0 by 1;
r0=lshift r0 by 1;   dm(ECPP)=r0;
r0=dm(m0,i0);
/* If the overlay section will end on an even address,
   save the last 16-bits to restore*/
r4=r4+r0; btst r4 by 0; if not sz jump save_internal_count;
r3=fext r4 by 0:16; r4=r4-r3;   r3=r3*r2 (ssi); r3=lshift r3 by -1;
r4=r4 or fdep r3 by 0:16;
r4=lshift r4 by 1;  r4=r4+1; dm(end_addr_stack)=r4;
i6=r4; r4=dm(0,i6); dm(end_addr_stack+1)=r4;
save_internal_count:
r0=r0*r2 (ssi); r0=r0+1; r0=lshift r0 by -1;   dm(ICPP)=r0;

r0=1;           dm(EMPP)=r0;
dm(IMPP)=r0;

//Determine if the external memory is 8- or 16-bit width
r6=lshift r6 by -1;
r0=dm(ICPP);
r3=dm(ECPP);
r0=lshift r0 by 1;
comp(r0,r3);
if lt jump external8;

//Set up for 16-bit external memory
external16:
r0=dm(ICPP);
r0=r0+r6;
dm(ICPP)=r0;
r0=dm(EIPP);
r0=r0-r6;
dm(EIPP)=r0;
r6=lshift r6 by 1;
r3=r3+r6;
dm(ECPP)=r3;
ustat1=PPBHC|PPDUR4|PP16;
dm(PPCTL)=ustat1;
jump startdma;

//Set up for 8-bit external memory
external8:
r0=dm(ICPP);
r0=r0+r6;
dm(ICPP)=r0;
r6=lshift r6 by 1;
r0=dm(EIPP);
r0=r0-r6;
dm(EIPP)=r0;
r6=lshift r6 by 1;
```

```
r3=r3+r6;
dm(ECPP)=r3;
ustat1=PPBHC|PPDUR4;
dm(PPCTL)=ustat1;

startdma:
bit set ustat1 PPEN|PPDEN;
dm(PPCTL)=ustat1;
nop;nop;nop;
/*  Wait for DMA to complete.  Note that, in this example's
    code, the DMA may not complete if another interrupt fires
    before the DMA's completion.  If this is consideration
    in your system, be sure to add this check to this code.  */
dma1_wait:      idle;

    //-------- Wait for PPI DMA to complete using polling------
    r0=dm(PPCTL);
    btst r0 by 17;
    if not sz jump (pc,-2);

/*  Restore register values from stack  */
m7=-1;
r0=dm(i7,m7);   l0=r0;
r0=dm(i7,m7);   m0=r0;
r0=dm(i7,m7);   i0=r0;
r0=dm(i7,m7);   i6=r0;
ustat1=dm(i7,m7);
r6=dm(i7,m7);
r5=dm(i7,m7);
r4=dm(i7,m7);
r3=dm(i7,m7);
r2=dm(i7,m7);
l8=dm(i7,m7);
i8=r1;
m8=0;

/*  Flush the cache.  If an instruction in previous overlay  */
/*  had been cached, it may be executed instead of the       */
/*  current overlays instruction.  (If pm transfers align.)  */
flush cache;

//Restore the saved 16-bit words corrupted by 32-bit to 48-bit conversion
r0=dm(start_addr_stack);
r1=0;
comp(r1,r0);
if eq jump check_end_stack;
r1=i6;
i6=r0;
r0=dm(start_addr_stack+1);
dm(0,i6)=r0;
i6=r1;

check_end_stack:
r0=dm(end_addr_stack);
r1=0;
comp(r1,r0);
if eq jump overlay_manager_end;
r1=i6;
i6=r0;
r0=dm(end_addr_stack+1);
dm(0,i6)=r0;
i6=r1;

overlay_manager_end:
r1=dm(i7,m7);
```

```
r0=dm(0,i7);

i7=dm(ov_stack);
m7=dm(ov_stack+1);
l7=dm(ov_stack+2);
nop;
/*  Jump to the location of the function to be executed.  */
jump (m8,i8) (db);
i8=r0;
m8=r1;

_OverlayManager.end:

/*****************************************************************************/
```

リスト1.SRAM 内に格納されたオーバーレイに対する基本的なオーバーレイ・マネージャ

```
/*****************************************************************************/
/*                                                                           */
/* File Name:  Ovl_Init.asm                                                  */
/*                                                                           */
/* Date:  August 29, 2003                                                    */
/*                                                                           */
/* Purpose:  This file runs through the ROM to look for where the overlays'  */
/* sections reside.  It then places the info for each section (live address, */
/* count size, and type) into respective buffers.  It also checks the types and */
/* accounts for how many words to increment in the ROM to look for the next  */
/* section's info.                                                           */
/*****************************************************************************/

#include "ovlay.h"
#include <def21262.h>

.extern num_ovl_sec;
.extern total_live_addr;
.extern total_sec_size;
.extern total_sec_type;
.extern read_data_bytes;

.section/dm dm_data;
.var scratch[3];

.section/pm pm_code;
.global _OvlInit;

_OvlInit:
                ustat1 = PPBHC|PPDUR23;
                dm(PPCTL)=ustat1;

                r0=0x1000600;
                dm(EIPP)=r0;
                M12=1;   //DAG2
                dm(EMPP)=m12;
                dm(IMPP)=m12;

                //Init index registers
                I9=total_live_addr;
                I10=total_sec_size;
                I13=total_sec_type;
                I14=num_ovl_sec;
```

```
// ========================  READ_BOOT_INFO ===================================
// Places TAG in R0, Internal Count in R2, and Destination Address in R3
// ---------------------------------------------------------------------------
read_boot_info: CALL read_prom_word;    // read first tag and count
                jump check_routine;

// ========================  read_prom_word  ==================================
// read_prom_word is a callable subroutine that DMA's in one 48-bit word from PROM
// It places the MS 32-bits in R3 and the LS 16 (right justified) in R2.
// For example, given the 48-bit word 0x112233445566,
//        R2 holds 0x00005566
//        R3 holds 0x11223344
// ---------------------------------------------------------------------------
read_prom_word: R0=scratch;
                DM(IIPP)=R0;           // 0x40004 = DMA destination address
                R0=3;
                DM(ICPP)=R0;
                R0=12;
                DM(ECPP)=R0;

                call read_data_bytes;

                R0=dm(scratch);                 // Copy TAG to R0
                RTS (DB);
                    R2=dm(scratch+1);    // Copy count to R2
read_prom_word.end: R3=dm(scratch+2);     // Copy address to R3



// ============================  CHECK_ROUTINE  ================================
// R0 holds the TAG, R2 holds the Word Count, and R3 holds the Destination Address
// Overlays have been mapped to a dummy live address in SDRAM space,
// Therefore, check for a destination address of greater than or equal to 0x600000
// in R3.  To advance the count by which to increment readings from the ROM, check
// each tag info and determine whether it needs to be incremented by 12 words, 6
// words, or none.
// Also check for final init.  When final init tag comes across, then the
// end of program is coming and there are no more overlays.
// ---------------------------------------------------------------------------

check_routine:
R9=PASS R0;             // check TAG

IF EQ JUMP final_init;  r9=r9-1;          // jump if fetched tag was 0, else tag--
IF EQ JUMP ZERO_init;  r9=r9-1;           // jump if fetched tag was 1, else tag--
IF EQ JUMP ZERO_init;    r9=r9-1;         // jump if fetched tag was 2, else tag--
IF EQ JUMP Internal_16;    r9=r9-1;       // jump if fetched tag was 3, else tag--
IF EQ JUMP Internal_32;    r9=r9-1;       // jump if fetched tag was 4, else tag--
IF EQ JUMP Internal_48;    r9=r9-1;       // jump if fetched tag was 5, else tag--
IF EQ JUMP Internal_64;    r9=r9-1;       // jump if fetched tag was 6, else tag--
IF EQ JUMP initliveinfozeros;   r9=r9-1;  // jump if fetched tag was 7, else tag--
IF EQ JUMP initliveinfozeros;  r9=r9-1;   // jump if fetched tag was 8, else tag--
IF EQ JUMP initliveinfo;   r9=r9-1;       // jump if fetched tag was 9, else tag--
IF EQ JUMP initliveinfo;  jump(pc,0);     // jump if fetched tag was A, else it
                                          // was an invalid TAG, so trap for debug

initliveinfo:
    R8 = OVLY_LIVE_START;
    R7 = R3-R8;
    if lt jump Internal_32;
    R8 = OVLY_LIVE_END;
    R7 = R3-R8;
```

```
    if gt jump Internal_32;
    r12=dm(EIPP);
    PM(I9,M12) = R12;       //write ROM live address to total_live_addr buffer
    PM(I10,M12) = R2;       //write word count to total_sec_size buffer
    PM(I13,M12) = R0;

    jump Internal_32;

initliveinfozeros:
    R8 = OVLY_LIVE_START;
    R7 = R3-R8;
    if lt jump ZERO_init;
    R8 = OVLY_LIVE_END;
    R7 = R3-R8;
    if gt jump ZERO_init;
    PM(I9,M12) = R12;   //write ROM live address to total_live_addr buffer
    PM(I10,M12) = R2;       //write word count to total_sec_size buffer
    PM(I13,M12) = R0;
    jump ZERO_init;

//no increment needed
ZERO_init:
    R1=0;   // No data to skip
    R2=0;   // No data to skip
    jump Update_PROM_address;

Internal_16:
    R2=R2+1;        // If count is odd, round up to make it even
    R1=0xFFFFFFFE; // because the loader pads an extra word
    R2=R2 AND R1;  // when the count is odd
    R1=2;           // Count is in 16-bit words, so multiply by 2
    jump Update_PROM_address;

Internal_32:
    R1=4;   // Count is in 32-bit words, so multiply by 4
    jump Update_PROM_address;

Internal_48:
    R2=R2+1;        // If count is odd, round up to make it even
    R1=0xFFFFFFFE; // because the loader pads an extra word
    R2=R2 AND R1;  // when the count is odd
    R1=6;           // Count is in 48-bit words, so multiply by 6
    jump Update_PROM_address;

Internal_64:
    R1=8;           // Count is in 64-bit words, so multiply by 8
    jump Update_PROM_address;

Update_PROM_address:
    R0=dm(EIPP);
    R1=R1*R2(SSI);
    R0=R0+R1;
    DM(EIPP)=R0;
    jump read_boot_info;

final_init:
    rts;
_OvlInit.end:
```

リスト2.オーバーレイ情報を取得するための、フラッシュ内に格納されたデータの解析

```
/*****************************************************************************/
/*                                                                           */
/* File Name:  Ovl_Sec_Info.asm                                              */
/*                                                                           */
/* Date:  August 27, 2003                                                    */
/*                                                                           */
/* Purpose:  This file parses all the info collected in the Ovl_Init.asm file    */
/*           for each overlay.  It checks for the number of sections within each */
/*           overlay.  It also checks each overlay section's type and size.  It  */
/*           accounts for the overlay id that's included in the data for each    */
/*           overlay by adding 6 words to the live address.  It accounts for     */
/*           the overlay id by subtracting 1 from the count size of the overlay. */
/*           Then, the information is written to the respective address and size */
/*           buffers.                                                        */
/*****************************************************************************/

#include "ovlay.h"

#include <def21262.h>
.extern runWordSize;

.extern num_ovl_sec;
.extern read_base_addr;
.extern read_buffer_addr;
.extern read_buffer_length;
.extern read_data_bytes;
.extern total_live_addr;
.extern total_sec_size;
.extern total_sec_type;

.segment/pm pm_code;
.global _OvlSecInfo;

_OvlSecInfo:

/* This first section of code checks the number of sections there are within each
   overlay */

lcntr = NUMBER_OF_OVERLAYS;             //count of overlays in this project

I12 = runWordSize;
I10 = total_sec_size;
I13 = num_ovl_sec;
I9 = total_live_addr;

do check_ovl.end until lce;
check_ovl:  R12=0;       //R12 stores number of sections.  Initiate to 0 to start.

R9 = PM(I12,1);     //read the total overlay size
repeat_check:   R8 = PM(I10,1);     //read the individual section size

R12 = R12+1;            //increment the total number of sections by 1

R9 = R9-R8;                 //subtract section size from total overlay size
if gt jump repeat_check;    //if not equal, then there are more sections in this
                            //overlay.  repeat the check.
nop;nop;

check_ovl.end:  PM(I13,1)=R12;  /*if equal, then there are no more sections in this
                                  overlay write number of sections in each overlay
                                  to buffer */
rts;

/* The next section of code accounts for the 1 extra address in the total_live_addr
   buffer holding the address of the overlay id information and the 1 extra word
```

```
   count in the total_sec_size buffer. */

I10 = total_sec_size;   //pointer to beginning of section size buffer
R10 = 0x4;              //R10 holds loop counter.  In this case, 4 overlays
R4=0x6;                 //6 8-bit words in ROM to increment count by
remove_id:      R2 = PM(I13,M12);       //Read number of sections in overlay
M14 = R2;               //Set M14 to number of sections in overlay
R6 = PM(I10,M9);        //Read individual section size buffer
R3 = PM(I9,M9);         //Read live address buffer
R6 = R6-R5;             //Subtract the section size by 1 (account for ovl id)
R3 = R3+R4;             //Add live address by 0x6 (account for ovl id)
PM(I10,M14)=R6;         //Write back section size to buffer
PM(I9,M14)=R3;          //Write back live address to buffer
R10 = R10-1;            //Do this for all four overlays.
if ne jump remove_id;   //Repeat if not done for all four overlays.

done:       rts;
_OvlSecInfo.end:
```

リスト3.各オーバーレイの情報を生成するための、フラッシュ内に格納されたタグの解析

参考

[1] *Using Memory Overlays (EE-66).* March 1999. Analog Devices, Inc.

[2] *Using Code Overlays from ROM on the ADSP-21161N EZ-KIT Lite (EE-180).* December 2002. Analog Devices, Inc.

[3] *Using External Memory with Third Generation SHARC Processors and the Parallel Port (EE-220).* Rev 2. February 2005. Analog Devices, Inc.

[4] *Programming an ST M25P80 SPI Flash with ADSP-21262 SHARC DSPs (EE-231).* Rev 1. February 2004. Analog Devices, Inc.

ドキュメント改訂履歴

Revision	Description
Rev 2 – March 11, 2005 *by Brian M.and Divya S.*	Generalized the discussion to third generation SHARC processors. Title changed to *Code Overlays on the Third Generation SHARC Family of Processors*
Rev 1 –February 17, 2004 *by Brian M.*	Initial Release of *Code Overlays on the ADSP-2126x SHARC Family of DSPs*