# Engineer To Engineer Note EE-69

アナログデバイセズ DSP および開発ツールに関する技術文書

http://www.analog.co.jp/industry/dsp/

## リンカー記述ファイル(LDF)を使いこなす

2106x のメモリー空間に独自の割り当てでコードやデータを配置したいということはよくあります。

例えば、21065L は外部メモリー上の命令を実行することが出来ますが、速度の低下は大きくなります。そこで、煩雑に実行するコードは内部メモリーに、あまり使わないコードは外部メモリーにおきたいところです。

アセンブリコードにしろ、C 言語のルーチンにしろ、再配置の方法は同じです。コードを任意の場所に配置するにはリンカー記述ファイル（LDF）を使います。

このアプリケーションノートは LDF の機能を説明し、例を使ってその使い方を紹介します。特殊な用途のアプリケーションのための複雑な LDF（例えば外部メモリー上のコードの実行など）はこの文書では取り上げません。

---

LDF を理解する第一歩は、DSP のプログラムを作るのに必要なファイルを理解することです。

### ソース・ファイル

はじめにソース・ファイルが必要です。これらのファイルは C 言語やアセンブリ言語で記述されます。こういったファイルをコンパイルあるいはアセンブルすることからプログラムのビルドが始まります。アセンブラの出力はオブジェクト・ファイルと呼ばれます（コンパイラの出力はアセンブリ・ファイルですがこれはアセンブラに入力されます）。VisualDSP のオブジェクト・ファイルは ”.DOJ ”という拡張子を持っています(このオブジェクト・ファイルは普通 debug サブディレクトリに出力されます)。

### オブジェクト・ファイル

コンパイラとアセンブラによって生成されたオブジェクト・ファイルはいくつかのセクションあるいはセグメント（オブジェクト・セグメントと呼びます）に分割されます。これらのオブジェクト・セグメントはコンパイルされたソース・コードの特定の部分を持っています。例をあげると、オブジェクト・セグメントはプログラムのオペ・コード（48 ビット幅）を持つことがあります。また、変数などのデータ（16,32,40 ビット幅）を持つこともあります。オブジェクト・セグメントの中にはこの文書とは無関係のものもあります（デバッグ情報などアプリケーションに無関係なものです）。

異なるオブジェクト・セグメントは異なる名前をもちます。この名前はソース・コードに記述します。ソースが C 言語かあるいはアセンブリ言語かでオブジェクト・セグメント名の宣言の仕方が異なります。

アセンブリ言語によるソースでは".section segName"の後ろで宣言されたコードやデータは"segName"と名づけられたオブジェクトセクションに配置されます。以下に例を示します：

```
.section/dm asmdata;
   .var foo[3];

.section/pm asmcode;
   r0=0x1234;
   r1=x4567;
   r2=r0+r1;
```

ここでオブジェクト・セグメント"asmdata"は配列 foo を持ちます。また、3 行からなるコードはオブジェクト・セグメント"asmcode"からなります。

C 言語のソース・コードではプログラマは segment( "segName" )命令を使うことが出来ます。例を挙げます：

```
segment(“ext_data”) int temp;

segment( "extern" ) void func1(void)
{
   int x=1;
}

void func2(void)
```

```
{
   int i=0;
}
```

このコードがコンパイルされると、func1 から生成されたコードは.DOJ ファイルの中の"extern"と名づけられた独自のオブジェクト・セグメントに格納されます。

では func2 はどうでしょうか？オブジェクト・セグメント名が指定されていない場合、コンパイラは既定値の名前を使います。つまりこの場合オブジェクト・ファイルは関数 func2 のコードを既定値のオブジェクト・セグメント"seg_pmco"に格納します。

*夫々のプロセッサごとのセグメント名の規定値は、対応するコンパイラのマニュアルを参照してください。SHARC の場合、コンパイラ・マニュアルの"C/C++ Runtime Model"にその記述があります。*

(注意：アセンブリ言語のプログラムには規定値のオブジェクト・セグメント名はありません。必ず.section で指定してください)

このオブジェクト・セグメント及びオブジェクト・セグメント名がどのようにメモリーへの配置に関与するかについては後述します。

**実行ファイル**

ひとたびソースファイルがアセンブルあるいはコンパイルされて*オブジェクト・ファイル*に変換されると、次はリンカーの仕事です。リンカーは*オブジェクト・ファイル*を一まとめにして*実行ファイル*にします。

*オブジェクト・ファイル*同様、実行ファイルは異なる"セグメント"に分割されています(これは DXE ファイルが採用している ELF フォーマットで説明されていることです)。このセグメントを DXE セグメントと呼びます。多分お気づきだと思いますが、DXE セグメントは DXE セグメント名を持ちます。

さて決定的に重大なことですが、DXE セグメント名はオブジェクト・セグメント名からは独立な存在です。夫々の名前空間は異なっており、つまりオブジェクト・セグメント名と同じ DXE セグメント名をつけることが出来ます(恥ずかしい話ですが、配布されているほとんどの例題プログラムは、同じ名前を両者につけています。これは論理的には問題ないとはいえ、間違えやすい悪いプログラミング・スタイルです)。

もう一つ、DXE の役割を理解するのも重要なことです。DXE は DSP にロードするものでなければ、ROM に焼きこむものでもありません。これは DXE には実コードや実データに加えて他の情報も入っているからです。これらの追加情報はローダーやデバッガと言った下流のツールが正しくデータを配置するために利用します。

**LDF はどのようにしてファイルを再配置するのか**

リンカーの仕事は言ってしまえば単純です。リンカーは LDF の中の命令に基づき、夫々のオブジェクト・ファイルからオブジェクト・セグメントを取り出します。そして LDF に記述してあるメモリーモデルに基づき、これら取り出したものを一つの DXE ファイルにまとめ上げます。

ではこれがどんな風に行われるのかサンプルの LDF を見てみましょう。ここではアセンブリ言語だけで書かれたプログラムをリンクするための LDF を見てみます。これは単純な例で、C 言語用ともなるとサポートライブラリをリンクするための記述が LDF に必要になります。

*このアプリケーションで使うプログラム 1 について説明しておきます。LDF は test.ldf です。番号がついたセクションは LDF の機能を説明するために使われます。セクションの番号は LDF のなかに記述する番号と対応します。*

1) LDF の中で最初に興味を引くのは Memory{}コマンドです。これはメモリーモデルを定義します。これによってリンカーにどこにメモリーが存在しを指示し、メモリー空間名をつけます。先ほどの繰り返しになりますがメモリー空間名はこれまでの名前空間とは別に存在し、オブジェクト・セグメント名や DXE セグメント名とは無関係です。LDF の重要な機能はこれら別々の空間の名前を関連付ける操作です。

2) 二つ目の、そしてもっとも重要な LDF の機能は Sections()部分でもたらされます。これこそがリンカーが本当の仕事を行う部分です。Sections()命令に与えられた三つの引数に基づいてリンカーは**オブジェクト・セグメントを DXE セグメントに与え、さらに DXE セグメントをメモリー空間に配置します。**

例で説明しましょう。test.ldf では Sections()命令の直下にある行は"isr_tbl"という名前のオブジェクト・セグメントを"dxe_isr"という名前の DXE セグメントに置いています。そうして最後にこの DXE セグメントを"mem_isr"という名前のメモリー空間に配置しています。これで DXE ファイルがデバッガで DSP にロードされる場合や、あるいは実際に ROM に焼かれてブ

ートする場合、これらの下流ツールはどの DXE セグメントがどの領域に配置されるべきかを知ることが出来ます。

3) 些細なことですが、われわれは"$OBJECT1"を"main.doj"と等価として扱っています。これは"$"で始まる記号は LDF ファイルではマクロとして扱われるからです。C 言語における#define 文と同じです。

この例ではそれほどありがたみがわからないものの、マクロ定義はオブジェクト・ファイルが複数あるような場合に非常に有用です。例えば複数のファイルの全ての seg_pmco オブジェクト・セグメントを同じメモリー・セグメントに配置したいような場合には二つの方法があります。一つの方法は全てのオブジェクト・ファイルを列挙する方法です。

```
dxe_pmco{
   INPUT_SECTIONS(
            main.doj(seg_pmco)
            config.doj(seg_pmco)
            dsp.doj(seg_pmco)
   )
} > mem_pmco
```

二つめの方法はマクロを使う方法です。

```
$ALL=main.doj, config.doj, dsp.doj;
```

そうして Sections()部の中で次のように使います：

```
dxe_pmco{
   INPUT_SECTIONS(
            $ALL(seg_pmco)
   )
} > mem_pmco
```

二つの方法は等価です。どちらの場合もリンカーは全てのオブジェクト・ファイルの中から"seg_pmco"オブジェクト・セグメントを探しだし、見つかったものを"dxe_pmco" DXE セグメントに詰め込みます。そうしてこのセグメントを"mem_pmco"メモリー空間に配置します。

注意：マルチプロセッサシステムの夫々のプロセッサは独自の"Processor"セクションを持ちます。したがって夫々のプロセッサは独自の"Sections()"宣言を持ちます。

4) LINK_AGAINST()コマンドはマルチプロセッサー構成のシステムを組むときに利用します。この命令を使うのはコードや命令を含む共有資源がある場合です。コマンドの使い方はこの文書の範疇を逸脱します。VisualDSP のリンカー・マニュアルを参照してください。

5) LDF の中の他の興味を引く点として OUTPUT()コマンドがあります。これは単に DXE ファイルの名前を指

```
ARCHITECTURE(ADSP-21065L)
SEARCH_DIR( $ADI_DSP¥21k¥lib )
[3:] $OBJECT1 = main.doj;
[1:] MEMORY
    {     mem_isr { TYPE(PM RAM) START(0x00008000) END(0x000080ff) WIDTH(48) }
          mem_pmco { TYPE(PM RAM) START(0x00008100) END(0x000087ff) WIDTH(48) }
          mem_pmda { TYPE(PM RAM) START(0x00009000) END(0x00009fff) WIDTH(32) }
          mem_dmda { TYPE(DM RAM) START(0x0000c000) END(0x0000dfff) WIDTH(32) }
    }
PROCESSOR p0
    {
[4:]        LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST )
[5:]        OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
[2:]        SECTIONS
          {         dxe_isr { INPUT_SECTIONS($OBJECT1(isr_tbl)) } > mem_isr
                    dxe_pmco { INPUT_SECTIONS($OBJECT1(seg_pmco)) } > mem_pmco
                    dxe_pmda { INPUT_SECTIONS(main.doj(seg_pmda)) } > mem_pmda
                    dxe_dmda { INPUT_SECTIONS(main.doj(seg_dmda)) } > mem_dmda
          }
}
```

定します。マクロ$COMMAND_LINE_OUTPUT_FILE が出力ファイル名として指定された場合、リンカーは DXE の名前を IDE が受け取ったプロジェクト名にします。それ以外の場合は単にファイル名を書いてください。たとえば次のようになります

```
OUTPUT(myfile.dxe)
```

---

英語の EE Note ページには C 言語対応版の LDF を含むサンプル ZIP ファイルがあります。これは二つのファイルに分かれた 3 つの関数の例で、コードがファイル単位あるいは SEGMENT() 指令によってどのように配置するかを例示しています。

(訳 S.H. 2001/10/30)

---

訳注：原文は内容が古いため .section ではなく .segment が使われています。この文書では適宜新しい記述になおして訳出しました。