フラッシュローダ開発ガイド

IAR Embedded Workbench®用

このガイドには、IAR Embedded Workbench におけるフラッシュローダメカニズムが記述されています。IAR Embedded Workbench におけるフラッシュローダの使用方法に関する情報を得ることができます。ガイドには、ユーザ専用フラッシュローダの開発やデバッグ方法も解説されています。最後に、フラッシュローダフレームワーク API ファンクションが詳細に解説されています。

フラッシュローダフレームワーク　バージョン 2

目次

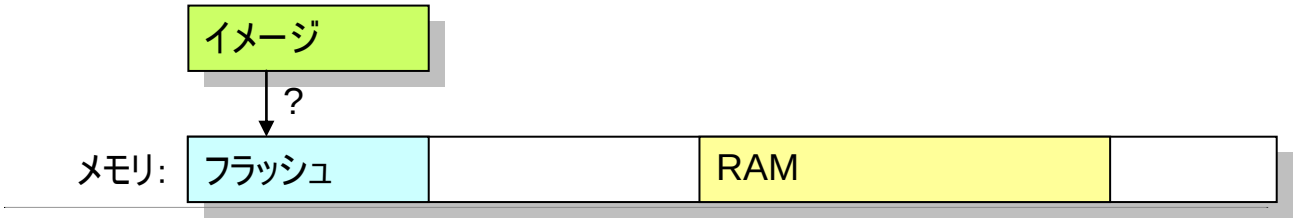
はじめに

開発ボードの多くは、主コードメモリとしてフラッシュメモリを使用します。通常、プログラムをダウンロードおよびデバッグする場合、フラッシュメモリは C-SPY から直接書き込むことができないため、ターゲットシステムで実行する「*フラッシュローダ*」と呼ばれる専用プログラムによりフラッシュやプログラミングを実施することができます。

注: フラッシュローダメカニズムは、デバッガによって直接書き込みできず、ターゲットシステムで実行する専用プログラムによって書き込む必要があるようなメモリの種類に有効です。フラッシュローダは、そのほとんどがフラッシュメモリで使用しますが、例えば、さまざまな種類の外部 RAM やディスクタイプのストレージデバイスでも使用することができます。しかしながら、このガイドでは、フラッシュメモリをメモリと仮定して取り扱います。

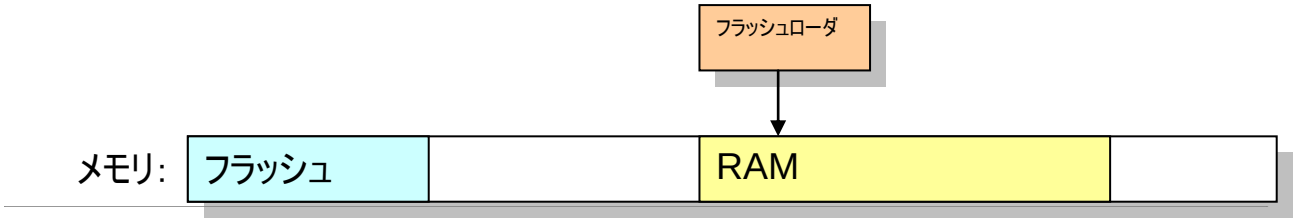
処理の概要

問題点：アプリケーションイメージをフラッシュメモリへダウンロードする必要がありますが、C-SPY はデータを直接、RAM にダウンロードすることしかできません。



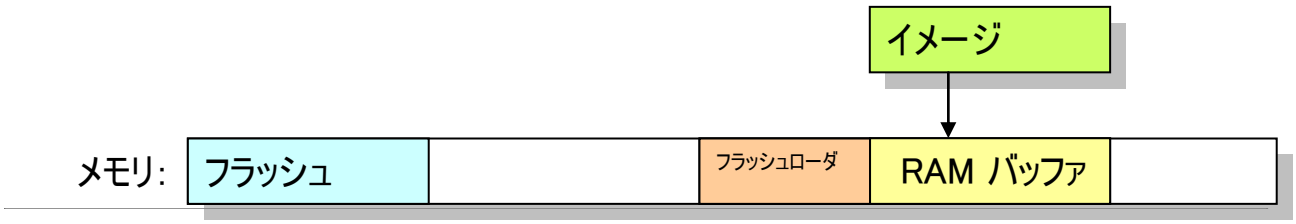
この問題を解決するために、次ぎの手順を実施します。

1 専用フラッシュローダを RAM へダウンロードします。

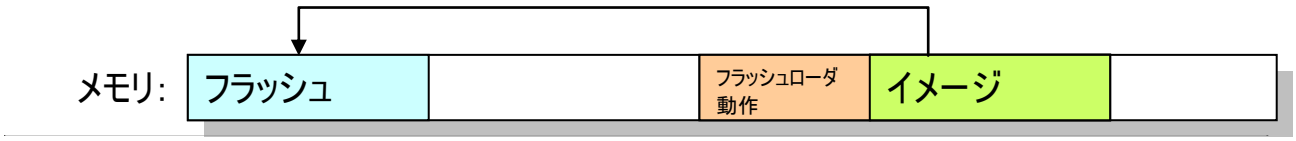


RAM の一部の領域も、ダウンロードバッファ用に確保されます。

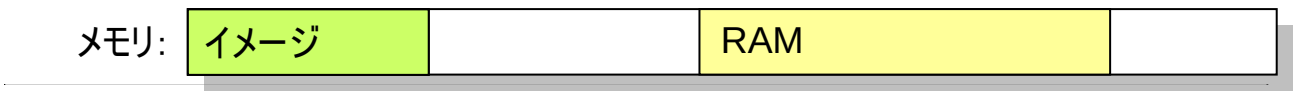
2 イメージが RAM バッファへ書き込まれます。



3 C-SPY によりフラッシュローダが起動されます。フラッシュローダは、RAM バッファからデータを読み取り、フラッシュメモリをプログラムします。



4 イメージがフラッシュメモリに書き込まれ、起動することが可能になりました。以降、フラッシュローダと RAM バッファは不要になり、RAM はすべてフラッシュメモリのアプリケーションに利用できます。



実際には、このプロセスはもう少し複雑になります。例えば、RAM バッファは通常、ダウンロードされるイメージよりも小さいため、フラッシュプログラミングを繰り返し実施する必要があります。

フラッシュローディングコンポーネント

フラッシュローディングを使用する際、その中には主要なコンポーネントが二つあります。

- フラッシュローダプログラム（略称は*フラッシュローダ*）
- フラッシュローダ構成ファイル（フラッシュメモリ*デバイス*構成ファイルとフラッシュメモリ*システム*構成ファイル）

フラッシュローダ

フラッシュローダは通常やや小さめのプログラムで、フラッシュメモリデバイスやデバイスファミリをプログラムすることができます。フラッシュローダは、複数ファンクションの小規模なセットで構成されており、これらは主としてフラッシュメモリの指定部分の消去や書き込みを目的とします。C-SPY はこのプログラムを RAM へダウンロードします（これは RAM のアドレスへリンクする必要がある）。プログラムを動作させるために、C-SPY は、フラッシュローダ内の関数の一つに PC をセットして、そのファンクションに対応するデータと命令を RAM バッファへ書き込み、実行します。関数が戻ると、プログラムはブレークポイントにヒットします。次に、C-SPY は関数の実行が終了したことを認識し、フラッシュローディングプロセスを続けるためにコールの実行をさらに進めることができます。すなわち、C-SPY がフラッシュローダ内の関数をコールすることになります。

フラッシュメモリデバイス構成ファイル

フラッシュメモリデバイス構成ファイルは、XML ファイル（ファイル名の拡張子 flash）で、特定フラッシュデバイスの C-SPY 関連プロパティすべてを記述します。例としては、フラッシュメモリの基準アドレスさらにはブロックおよびページサイズなどの詳細などがあります。ファイルは、どのフラッシュローダを使用するかも指定します。

フラッシュメモリシステム構成ファイル

フラッシュメモリシステム構成ファイルは、XML ファイル（ファイル名拡張子 board）で、C-SPY に対して完全な開発ボードのフラッシュローディングプロパティを記述しています。いくつかの*パス*で個別にプログラムする必要があるフラッシュメモリを複数種類、ボードが持っている場合、このファイルは、（最適な.flash ファイルを介して）複数のフラッシュデバイスに対するリファレンスを含むことも可能です。この場合、.board ファイルは、さまざまなフラッシュデバイスに属するイメージの具体的なアドレス範囲を指定することもあります。

一つの.board ファイルは、特定のボードでフラッシュローディングを実行するのに必要な情報を完全に指定します。そのようなファイルは、様々な開発ボードに対して事前に準備することができ、さらに IAR Embedded WorkbenchIAR IDE のプロジェクト**[Options]**（**オプション**）ダイアログボックスで作成、あるいは修正することができます。

.board ファイルは、特定ボードのフラッシュメモリに関するすべての情報をかならずしも記述するわけではなく、むしろ、与えられた IAR Embedded Workbench プロジェクトでボードを使用するのに必要なものをすべて記述していることに留意してください。例えば、ボードが二つのフラッシュデバイスを含んでいて、一つはブートローダでもう一つはアプリケーションに使用される場合、その中の一つのみが、与えられたプロジェクトに関係していると想定されます。したがって、各プロジェクトに一個ずつ、二種類の.board ファイルが存在することになります。

フラッシュメモリの概念

さまざまなフラッシュデバイスをサポートするために、C-SPY は、フラッシュデバイスの特性を詳述するいくつかの概念を利用しています。

ページ

ページは、フラッシュメモリにおける最小の書き込み可能単位です。多くのフラッシュデバイスは、一回の書き込み操作で、例えば 128 または 256 バイト未満の書き込みをすることができません。C-SPY が、1 ページよりも小さい書き込みをフラッシュローダに対して要求することはなく、ページを埋めるために必要に応じてパディングを使用します。もちろんのこと、そのような制限が無く、1 バイトのページサイズを指定できるフラッシュデバイスもあります。

ブロック

ブロックは、フラッシュメモリにおける最小の消去可能単位です。例えば、256 バイトのページサイズを持つフラッシュデバイスが、4K バイト単位でフラッシュメモリを消去する必要があるかもしれません。ブロックサイズは、常にページサイズの倍数でなければなりません。フラッシュデバイスは、異なるサイズからなる複数のブロックで構成することができます。また、そのような制限が適用されないこともあり、その場合、ブロックサイズはページサイズに等しくなりこともあります。

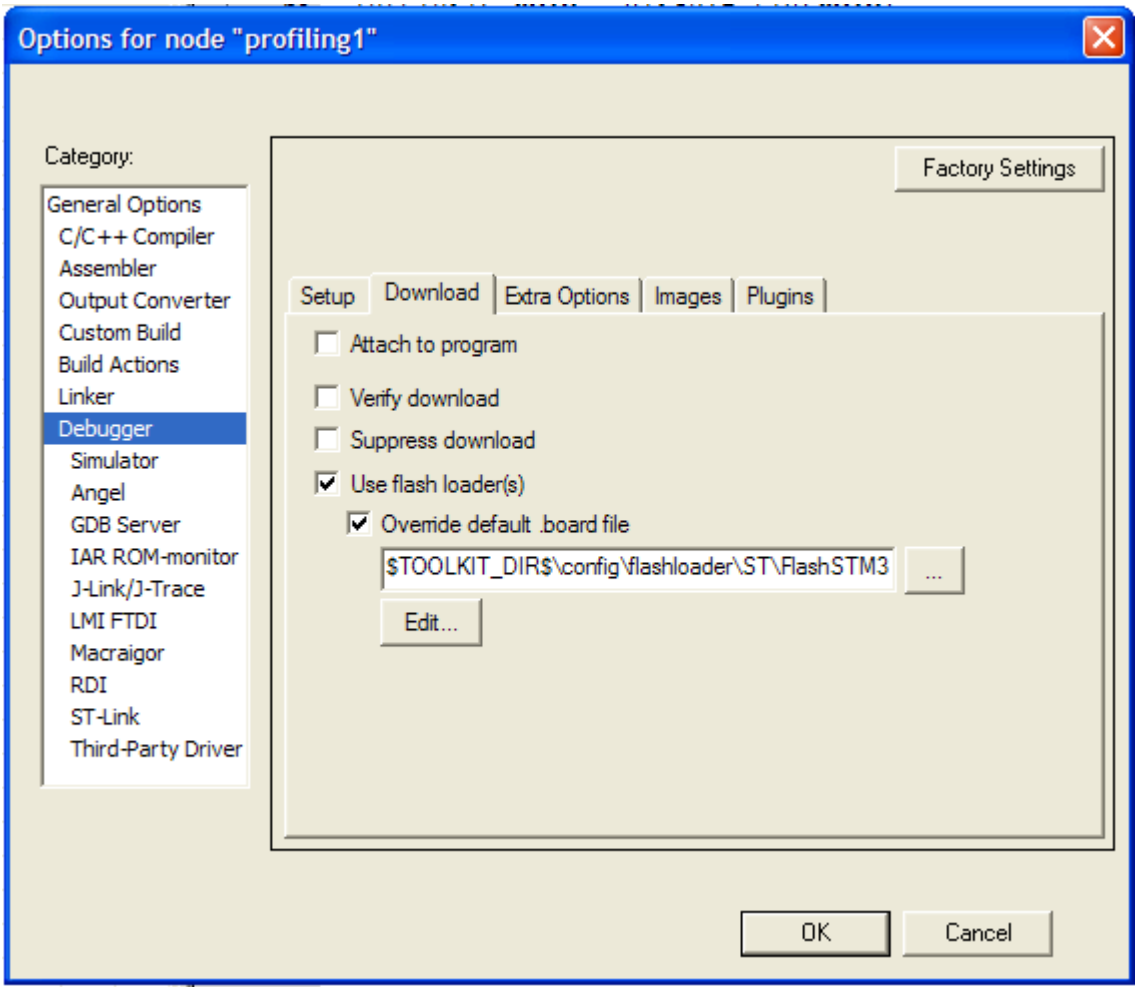
基準アドレス

これは、フラッシュメモリデバイスに*書き込みが行われるときの*、その開始アドレスです。いくつかのフラッシュメモリは、単に固定アドレス範囲に対応する記憶領域をもっており、その時の基準アドレスは、その範囲の開始アドレスになります。他のフラッシュメモリは、プログラミング、または後のアプリケーション実施の際、さまざまなアドレスへマッピングされます。したがって、このときの基準アドレスは、プログラミングが行われているときにメモリがマッピングされているアドレスになります。しかし、他のフラッシュメモリの中にはメモリマッピングが全く行われないものもあり、外部ディスク装置のように動作します。すなわち、基準アドレスは、単に、プログラミングが行われる際、メモリの開始位置に使用するのに最適なアドレスとなります。

C-SPY の視点から見ると、フラッシュメモリデバイスは、与えられたアドレスから開始し、ブロックシーケンス（異なるサイズも可能）で構成され、各々は複数のページ数で構成されています。ブロックシーケンスには、ギャップを含めることも可能です。

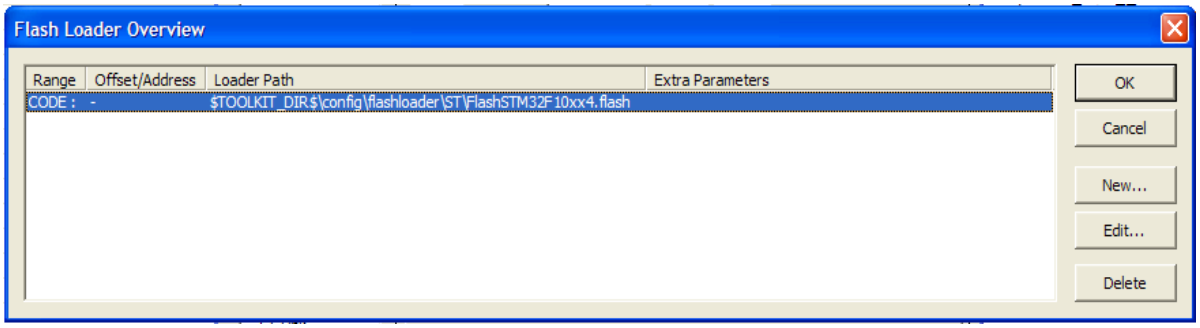
フラッシュローダの使用

5 IAR Embedded Workbench IDE を使用してフラッシュローディングを作動するには、**[Project]>[Options]>[Debugger]>[Download](プロジェクト>オプション>デバッガ>ダウンロード)**を選択します。



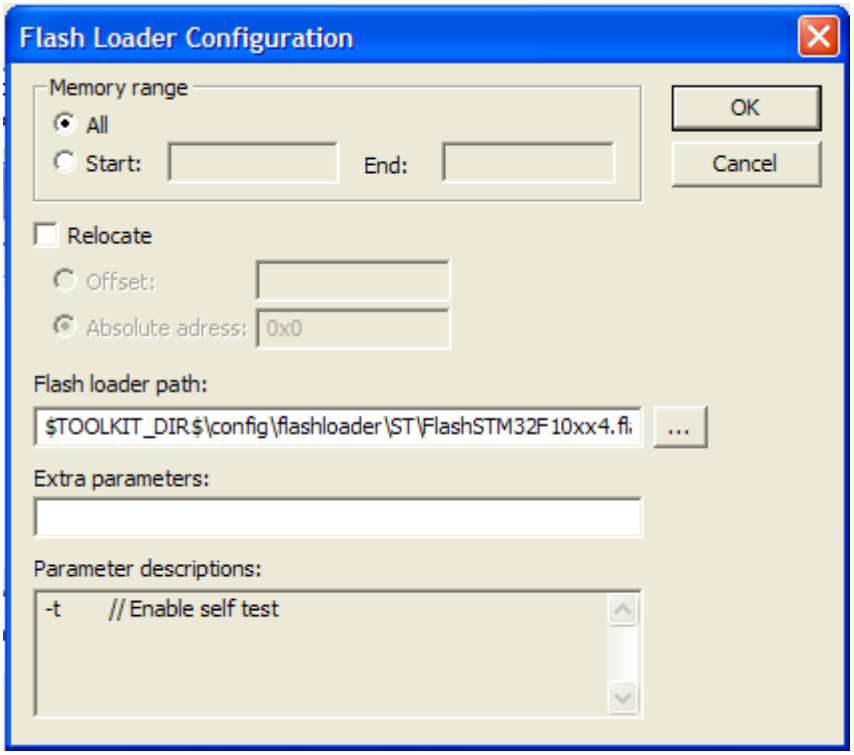
[Use flash loader(s)](フラッシュローダの使用) と**[Override default board file](デフォルトボードファイルのオーバーライド)**オプションを選択して、フラッシュメモリシステム構成ファイル(.board)を指定します。ブラウズボタンを使用して、あなたのシステムに最適な事前定義されたファイルを選択します。そのようなファイルが存在しない場合、新しいファイルを作成するか、**[Edit](編集)**ボタンをクリックして既存のファイルを修正できます。

6 **[Flash Loader Overview](フラッシュローダ概要)**ダイアログボックスが表示されます。IAR Embedded Workbench ディレクトリに置かれた事前定義されたファイルの 1 つを編集する場合、引き続き、修正されたファイルを異なるディレクトリに保存するように促されますので、注意してください。



ダイアログボックスには、ボード上の各フラッシュメモリデバイス、または各フラッシュローダパスに対する情報が一列に表示されます。**[New](新規)**をクリックして新しいパスを定義するか、**[Edit](編集)**をクリックして既存のパスを修正するか、あるいは**[Delete](削除)**をクリックしてリストからパスを削除します。

7 **[New](新規)**または**[Edit](編集)**ボタンをクリックして**[Flash Loader Configuration](フラッシュローダ構成)** ダイアログボックスを表示します。



[Memory range](メモリ範囲)セクションは、このパスに対してフルデバッグファイルのどのサブセットを使用するかを指定します。**[Relocate](リロケート)**セクションは、フラッシュをプログラミングする前にデバッグファイルデータの選択可能なリロケーションを指定します。**[Flash loader path](フラッシュローダパス)**テキストフィールドは、この特定フラッシュメモリデバイスに対して.flash ファイルを指定します。**[Extra parameters](エキストラパラメータ)** テキストフィールドは、フラッシュローダへ渡されるべき、スペースで区切られた引数(すなわち、FlashInit ファンクションに対する argc/argv パラメータ)を含んでいます。**[Parameter descriptions](パラメータ記述)** テキストフィールドは、存在していれば使用できる可能性のあるエキストラパラメータに関する情報を表示します。

終了したら、**[OK](確認)**をクリックします。

8 結果は、一個の.board ファイルになり、これにより完全なフラッシュローディングシーケンスが指定されます。

フラッシュプログラミングプロセスの詳細

フラッシュローダにおける二つの最も重要なファンクションは、FlashWrite および FlashErase と呼ばれています。前者は、RAM バッファからフラッシュメモリへ多量のバイト数(常にページ数全体)のデータを書き込んだり、またはコピーしたりします。後者は、フラッシュメモリブロックを一個消去します。イメージファイルのデータを使用しながら、C-SPY は、データを RAM バッファへ繰り返し書き込み、さらにフラッシュローダの FlashWrite 機能を呼び出します。この時、以下の制限があります。

- 書き込みはシーケンシャルに行われ、最も低いアドレスから開始
- バッファは、常に整数のページ数を含む
- ページがデータによって満たされていない場合はかならず、バッファでパッディングが実施される

- 与えられたブロックの最初のページが書き込まれる前に、FlashErase ファンクションが呼び出されブロック全体が消去される。

詳細は以下の通り。

1 フラッシュメモリへダウンロードされる予定のアプリケーションはイメージファイルに存在します。.board ファイルは C-SPY によって読み出され、一個または複数のフラッシュローディングパスを指定し、それらはボード上の各フラッシュメモリデバイスに対応します。

2 各パスに対して、オリジナルイメージファイルの具体的なアドレス範囲（またはサブセット）が指定されます。それに応じて、イメージファイルは各パスに対する個別イメージファイルに分割されます。パスが一個しかない場合、オリジナルのイメージファイルがそのまま使用されます。

3 各パスは、、具体的なフラッシュローダを指定するフラッシュデバイスファイル(.flash ファイル)を指定します。

4 C-SPY は、カレントパスにあるフラッシュローダ RAM へダウンロードします。

5 パスがオフセットを指定する場合、それに応じて、イメージファイルの全レコードはリロケートされます。

6 C-SPY は PC を FlashInit に設定（技術的には、FlashInit をこれから呼び出すであろうラベルに設定）します。

7 パラメータとデータが RAM バッファへ書き込まれます。

8 プログラム実行が開始されて、FlashInit が実行され、特別なブレークポイントに達すると C-SPY が制御を再開します。FlashInit は、.flash ファイルにあるいくつかの情報をオーバーライドする機会を持っており、それにはページサイズやブロックレイアウトが含まれます。

9 C-SPY は、フラッシュメモリのページやブロックレイアウト、および RAM バッファのサイズに最適な個数になるように、イメージファイルのデータを分割します。

10 あるブロックが最初に書き込まれる前に、ブロックはまず消去される必要があります。この場合、手順は次のステップへ続きます。そうでなければ、手順はステップ 13 へ続きます。

11 RAM パラメータは、ブロックのサイズとそのアドレスが割り当てられます。

12 C-SPY は PC を FlashErase にセットし、プログラム実行を開始します。実行が終了すると、プログラムはブレークポイントに達します。

13 C-SPY はデータのいくつかを RAM バッファへ書き込みます。

14 C-SPY は PC を FlashWrite に設定し、プログラム実行を開始します。実行が終了すると、プログラムはブレークポイントに達します。

15 さらにデータがある場合、手順はステップ 10 へ戻ります。

16 さらにパスがある場合、手順はステップ 3 へ戻ります。

17 最終的なアプリケーションに対応するデバッグ情報が読み込まれます。

18 C-SPY は PC を最終アプリケーションの開始アドレスへ設定します。

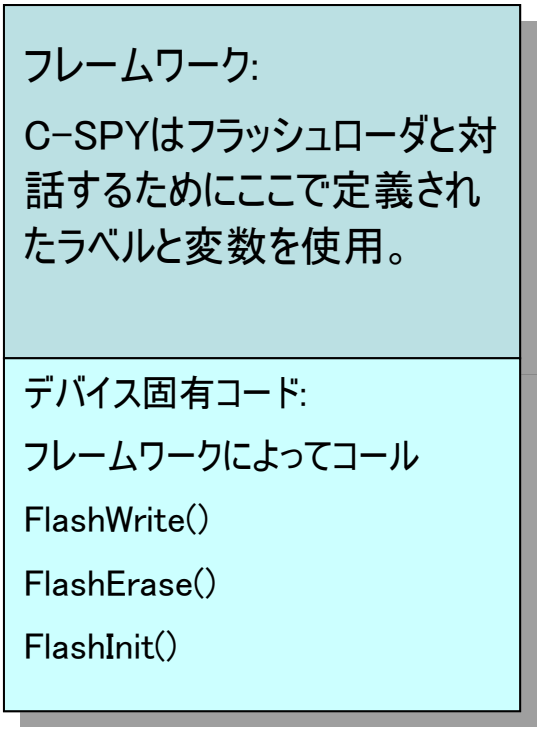
19 任意に、main で実行を開始します。

フラッシュローダの作成

ターゲットシステムに最適なフラッシュローダがない場合、あなた自身のフラッシュローダを開発することが出来ます。

フラッシュローダは、IAR Embedded Workbench を使用して開発できるネイティブアプリケーションです。

- フラッシュローダ*フレームワーク*ソースコードは IAR Embedded Workbench より供給
- デバイス固有ソースコード – 通常、C ファンクションのスモールセットで、ユーザがインプリメントする必要がある。



簡単な例

以下の例は、RAM バッファから宛先アドレスへ数バイトを単純にコピーするフラッシュローダの完全なソースコードを示して（フレームワークのソースコードは除く）、います。

```
#include "flash_loader.h"

uint32_t FlashInit(void *base_of_flash, uint32_t image_size,
uint32_t link_address, uint32_t flags)
{
return RESULT_OK;
}

uint32_t FlashWrite(void *block_start,
uint32_t offset_into_block,
uint32_t count,
char const *buffer)
{
char *to = (char*)block_start + offset_into_block;
while (count--)
  {
*to++ = *buffer++;
  }
return RESULT_OK;
}

uint32_t FlashErase(void *block_start, uint32_t block_size)
{
char *p = (char*)block_start;
while (block_size--)
  {
*p++ = 0;
```

```
 }
return RESULT_OK;
}
```

FlashWrite と FlashErase のパラメータは、FlashInit で与えられるフラッシュメモリ基準アドレスと連携して、プログラムされるフラッシュメモリ部分のアドレスを完全に指定します。したがって、同様のフラッシュプログラミングアルゴリズムを使用する限り、与えられたフラッシュローダは、合計サイズ、ページサイズ、ブロックレイアウトの異なる、さまざまなフラッシュデバイスで使用できます。フラッシュメモリ構成ファイル（.flash）は、フラッシュメモリデバイス間の変数を指定するために使用されます。

このドキュメントの巻末にある参考用セクションには、すべてのフレームワークファンクションについて詳細に解説されています。

フラッシュローダの構築

前述したように、フラッシュローダは、フレームワークソースコードとユーザが提供するデバイス固有ソースコードによって構築されています。以下のファイルはフレームワークの一部分で、あなたのフラッシュローダにも利用できるはずです。

flash_loader.c	フレームワークソースコード。このファイルは、あなたのフラッシュローダプロジェクトの一部分として利用できますが、修正せずそのままご利用ください。
flash_loader.h	フレームワーク宣言で、例えば、あなたのソースコードの C プロトタイプ。
flash_loader_extra.h	補助的なフレームワーク宣言で、ソースコードでまれに必要。
flash_loader_asm.s	ローレベルのプロセッサ固有フレームワークソースコード。マイクロコントローラに応じて、このファイルは、異なるファイル名拡張子を持つことがあるので注意してください。このファイルは、あなたのフラッシュローダプロジェクトの一部分として利用できますが、修正せずそのままご利用ください。
Template\flash_config.h	あなた自身の構成ファイルのテンプレートです。コピーをして、ファイルに適したガイドラインにしたがって、コピーしたものを編集してください。

ファイルのコピーやフラッシュローダの例は、インストレーションディレクトリの *target*\src\flashloader にあります。フラッシュローダを作成した際、考慮すべきいくつかの事柄があります。例えば、フラッシュローダは RAM のアドレスへリンクする必要があり、フラッシュローダは、main ファンクションのようなエントリーポイントを含んでいません。

フラッシュメモリデバイス構成ファイルの詳細

前述したように、フラッシュメモリデバイス構成ファイル（.flash）は、XML ファイルで、特定のフラッシュメモリデバイスのプロパティを指定し、その中には、プログラミングをするためにどのフラッシュローダを使用するのか等が含まれています。次は、ファイルの一例です。

```
<?xml version="1.0" encoding="iso-8859-1"?>

<flash_device>
<exe>$TOOLKIT_DIR$\config\flash\P8_family\flash_p8.out</exe>
<flash_base>0x20000</flash_base>
<page>256</page>
<block>2 0x100</block>
<block>3 0x200</block>
```

</flash_device>

ファイルは*エレメント*(必須とオプション項目あり)で構成され、各エレメントは*タグ*といくつかの*コンテンツ*で構成されます。

必須エレメント

exe	フラッシュローダへのパス指定します。パスは$TOOLKIT_DIR$のような引数変数を含められます。
flash_base	書き込みを実施するときのフラッシュメモリの基準アドレスを指定します。
page	フラッシュメモリのページサイズを指定します。
block	このエレメントを一個または複数個順番に使用して、フラッシュメモリのブロックレイアウトを指定します。各エレメントは、10 進のカウント値に続いて 16 進のブロックサイズを含みます。エレメントシーケンスは、フラッシュメモリのブロックシーケンスを完全に指定する必要があります。上の例では、フラッシュデバイスは、サイズ 0x100 の二つのブロックを含み、さらにサイズ 0x200 の三つのブロックが続いて、0x800 の合計サイズになります。

オプションエレメント

gap	指定されたブロックシーケンスには、block エレメントが混在した一つまたは複数の gap エレメントも含むことがあります。各 gap エレメントは、16 進形式のギャップサイズを含みます。ギャップサイズは、フラッシュメモリを含まないフラッシュメモリ領域を指定します。C-SPY は、ギャップ内に配置される可能性のあるイメージファイルのデータに対して、エラーを発行します。
macro	フラッシュローダのダウンロードと連携してロードされる C-SPY マクロファイルへのパスを指定します。三つの C-SPY マクロファンクションがあり、それらはマクロファイルに定義されている限り自動的に呼び出されます。 execUserFlashInit は、フラッシュローダをロードする直前に呼び出されます。 execUserFlashReset は、ローディングに続くリセットの直後に呼び出されます。 execUserFlashExit は、フラッシュローダがアンロードされる前で、フラッシュローディングの終了直後に呼び出されます。
filler	ページ境界へパディングの書き込み操作を実施する際に使用するバイト値を 10 進数で指定します。デフォルト値は 255(0xFF)です。
checks	このエレメントが 0 を含んでいる場合、フラッシュローダ機能(FlashWrite などの)からのエラー戻り値はチェックされません。このチェックを無効にすることで、パフォーマンスがわずかに向上します。経験上これらのチェックが不要であるとわかっている場合のみ、このエレメントを使用してください。
aggregate	このエレメントが 1 を含んでいる場合、C-SPY は、書き込み操作を一つ以上のブロックに結合させることで、RAM ダウンロードバッファをさらに効率良く使用することに努めます。これは、ブロックサイズが RAM バッファよりも著しく小さい場合、もしくはその場合に限って有効なパフォーマンスの最適化で、これにより少なくとも二つ(できればそれ以上)のブロックが、ダウンロードバッファに適合します。このエレメントでは、フラッシュローダが一回の操作で一つ以上のブロックをプログラムできるといことが必須になります。
args	FlashInit ファンクションに対する argc/argv の形式で、フラッシュローダへ引数を

渡します。パラメータは、*newline* 文字によって区切る必要があります。

args_doc　フラッシュローダ FlashInit ファンクションによって受けいれられるパラメータの記述を指定します。内容は、IAR Embedded Workbench IDE の**[Flash Loader Configuration](フラッシュローダ構成)** ダイアログボックスに表示されます。このエレメントは、*newline* 文字で区切られる複数行のテキストを含められます。

あるマイクロプロセッサは、さまざまなバリエーションで利用でき、各々は同じタイプのフラッシュメモリを装備していますが、異なるサイズやアドレス、場合によっては異なるブロックサイズを持っています。そのようなシナリオの場合、フラッシュローダは一個だけで十分ですが、.flash ファイルはいくつか必要になります。以下の仮想 *P8* プロセッサファミリに関するいくつかのバリエーションが記載された　表をご覧ください。

バリエーション	フラッシュサイズ (K バイト)	フラッシュ基準アドレス	フラッシュブロックレイアウト	構成ファイル
P8_1	1	0x10000	4 * 0x100	flash_p8_1.flash
P8_2a	2	0x10000	8 * 0x100	flash_p8_2a.flash
P8_2b	2	0x10000	4 * 0x200	flash_p8_2b.flash
P8_4a	4	0x10000	8 * 0x100 4 * 0x200	flash_p8_4a.flash
P8_4b	4	0x20000	16 * 0x100	flash_p8_bb.flash

この例では、五種類の.flash ファイルがありますが、各プロセッサのバリエーションは、同一のフラッシュプログラミングアルゴリズムを必要とする同じ種類のフラッシュメモリデバイスを装備しているため、各ファイルは同じフラッシュローダを指定しています。

フラッシュメモリシステム構成ファイルの詳細

前述したように、フラッシュメモリシステム構成ファイル(.board)は、XML ファイルで、フラッシュメモリデバイスに関して、特定の開発ボードのプロパティを指定します。ファイルは*エレメント*で構成され、各エレメントは*タグ*といくつかの*コンテンツ*で構成されます。最上位レベルで、ファイルは一つ以上の*パス*を記述し、各々は*パスエレメント*によって指定されます。このようなファイルの例を以下に示します。これは二種類のフラッシュメモリデバイスに対する二つのプログラミングパスを指定します。

```
<?xml version="1.0" encoding="iso-8859-1"?>

<flash_board>
<pass>
<loader>$TOOLKIT_DIR$\config\flash\flash_p8_2a.flash</loader>
<range>CODE 0x20000 0x207ff</range>
<abs_offset>0x10000</abs_offset>
</pass>
<pass>
<loader>$TOOLKIT_DIR$\config\flash\flash_p8_2b.flash</loader>
<range>CODE 0x20800 0x21000</range>
<abs_offset>0x10000</abs_offset>
</pass>
<ignore>CODE 0x22000 0x220ff</ignore>
</flash_board>
```

各パスエレメントは補助エレメントを含んでおり、以下のように必須エレメントとオプションエレメントがあります。

必須エレメント

loader	フラッシュメモリデバイス構成ファイル(.flash)へのパスを指定します。パスは$TOOLKIT_DIR$のような引数変数を含めることができます。

オプションエレメント

range	オリジナルイメージファイルのサブセットを指定し、これは特定のフラッシュメモリデバイスでプログラムする必要があります。このエレメントの内容には、セグメント名（通常）に続いて、範囲を示す最初と最後のバイトのアドレスが 16 進形式で入ります。フラッシュメモリデバイスが一個のみの場合、デフォルト範囲はイメージファイル全体の範囲になります。複数のパスが存在する場合、C-SPY がイメージをどう分割するかを知る必要があるため、エレメントは必須となります。
abs_offset	このエレメントは、リンカによってイメージが設置された場所のアドレスとは異なるアドレスのフラッシュメモリにイメージに書き込むために使用されます。例えば、フラッシュメモリデバイスが、プログラミング時にメモリ内のあるアドレスにマッピングされ、その後、実行時に別のアドレスに再マッピングされる場合、フラッシュメモリのプログラミング時、補正するために最適なオフセットを使用する必要があります。このエレメントは、絶対アドレスを指定し、ここにはイメージの最初のバイトを配置する必要があります。
rel_offset	このエレメントは、abs_offset に似ていますが、相対オフセットを指定します。これにより、フラッシュへ書き込む前にイメージファイル内の各レコードを移動する必要があります。オフセットは、正数と負数のどちらも使用できます。abs_offset の rel_offset の両方を同じパスで使用することが出来ないのでどちらか片方を使用します。
flash_base	書き込まれる際のフラッシュメモリの基準アドレスを指定します。このエレメントが含まれている場合、.flash ファイル内の対応するエレメントをオーバーライドします。
args	FlashInit ファンクションに対する argc/argv の形式で、フラッシュローダへ引数を渡します。パラメータは、*newline* 文字によって区切る必要があります。パラメータは、.flash ファイル内で指定される全パラメータに付加されます。フラッシュローダがパラメータを適切に処理する場合、これらのパラメータは、デバイス構成ファイルで指定されるパラメータをオーバーライドできます。
ignore	オリジナルイメージファイルのサブセットを指定し、このファイルはフラッシュローディングに依存*しない*ものでなければなりません。このエレメントは、セグメント名（通常 CODE）で構成され、さらに、範囲を示す最初と最後のバイトのアドレスが 16 進形式で続きます。エレメントは、例えば、オリジナルイメージの部分を RAM へダウンロードする場合、あるいは、その部分がいくつかの ROM にすでに存在している場合、使用できます。エレメントは、いくつかの範囲を繰り返し指定できます。このエレメントは、pass エレメントと同じレベルで指定されることに注意してください。

フレームワークの参照事項

以下のファンクションは、フラッシュローダによってインプリメントする必要があります。

FlashWrite（フラッシュ書き込み）

```
uint32_t FlashWrite(void *block_start,
uint32_t offset_into_block,
uint32_t count,
char const *buffer);
```

パラメータ:

block_start	この書き込み操作によって書き込まれるブロックの最初のバイトを指定します。
offset_into_block	この書き込み操作が開始するカレントブロックに対してどの位離れているかを指定します。書き込まれる最初のバイトの絶対アドレスは、block_start + offset_into_block です。
count	書き込むバイト数を指定します。
buffer	書き込むバイトが含まれるバッファを示すポインタです。

戻り値: RESULT_OK または RESULT_ERROR のいずれか。

FlashErase（フラッシュ消去）

```
uint32_t FlashErase(void *block_start,
uint32_t block_size);
```

パラメータ:

block_start	消去するブロックの最初のバイトを指定します。
block_size	ブロックのサイズをバイトで指定します。

戻り値: RESULT_OK または RESULT_ERROR のいずれか。

FlashInit（フラッシュ初期化）

```
#if USE_ARGC_ARGV
uint32_t FlashInit(void *base_of_flash, uint32_t image_size,
uint32_t link_address, uint32_t flags,
int argc, char const *argv[]);
#else
uint32_t FlashInit(void *base_of_flash, uint32_t image_size,
uint32_t link_address, uint32_t flags);
#endif;
```

上に示されるように、FlashInit には二種類のプロトタイプがあり、これらは、flash_config.h で指定しなければならないプリプロセッサマクロ USE_ARGC_ARGV の値によって決定されます。柔軟性を特に必要な場合、引数によって調整してください。実際の引数は、.flash ファイル、または IAR Embedded Workbench IDE のプロジェクト[**Options**](オプション)ダイアログボックスで指定できます。

パラメータ:

base_of_flash	フラッシュメモリ範囲全体の最初のバイトを指示します。
image_size	フラッシュメモリに書き込まれる全イメージのサイズをバイトで指定します。
link_address	オフセットの前、または、複数のパスがある場合、このパスに使用されるイメージサブセットの最初のバイトの前に、イメージの最初のバ

	イトのオリジナルリンクアドレスを指定します。フラッシュローダすべてが、このパラメータを必要とするわけではありません。
flags	オプションのフラッグを指定します。詳細については下記を参照してください。

戻り値：RESULT_OK または RESULT_ERROR のいずれかですが、別の戻り値も使用できます（詳細は下記を参照）。

高度な FlashInit ファンクション機能

FlashInit ファンクションは、フラッシュローダで呼び出される最初のファンクションです。そのため、このファンクションは、実際のフラッシュプログラミングが開始する前に、C-SPY に対して特別な情報を提供するための機会を提供します。追加情報を提供するには、.flash ファイルで指定されたプロパティをオーバーライドしなければなりません。オプションのヘッダファイル flash_loader_extra.h で定義されるマクロセットを使用してください。内部で、このファンクションはフレームワークで定義される構造体変数へのアクセスを必要としており、この変数は、C-SPY とフラッシュローダの間で交互に情報を受け渡すために使用されます。変数の名前は、theFlashParams で以下のようなヘッダファイルでも宣言されます。

```
typedef struct {
uint32_t base_ptr;
uint32_t count;
uint32_t offset_into_block;
void *buffer;
uint32_t block_size;
} FlashParamsHolder;

extern FlashParamsHolder theFlashParams;
```

ページサイズのオーバーライド

ページサイズをオーバーライドするには、SET_PAGESIZE_OVERRIDE マクロを使用して、この例のように、戻り値に SET_PAGESIZE_OVERRIDE ビットを設定します。

```
uint32_t FlashInit(void *base_of_flash, uint32_t image_size)
{
SET_PAGESIZE_OVERRIDE(128); // New page size
return RESULT_OK | OVERRIDE_PAGESIZE;
}
```

バッファサイズのオーバーライド

ダウンロードバッファサイズは通常、FlashBufferStart と FlashBufferEnd の二つのラベル位置によって決定され、それらの位置はリンク時間に取得されます。RAM サイズのみ異なる複数のデバイスに対して同じフラッシュローダを使用するために、フラッシュローダはバッファサイズをオーバーライドできます（フラッシュローダが、有効な RAM の実容量を決定できる場合）。SET_BUFSIZE_OVERRIDE マクロを使用し、FlashInit ファンクションからの戻り値に OVERRIDE_BUFSIZE ビットを設定します。バッファサイズを減少しないでください。

```
uint32_t FlashInit(void *base_of_flash, uint32_t image_size)
{
SET_BUFSIZE_OVERRIDE(0x1000); // New buffer size
return RESULT_OK | OVERRIDE_BUFSIZE;
}
```

ブロックレイアウトのオーバーライド

通常、.flash ファイルは、block および gap タグを使用して、フラッシュメモリのブロックレイアウトを指定します。フラッシュメモリデバイス自身に対してなんらかの方式でクエリーを実行することにより、フラッシュローダにブロックレイアウトを決定させることは、実際に役立つ方法です。フラッシュローダがレイアウトを指定

する必要がある場合、フラッシュローダバッファにレイアウトを記述し、FlashInit のリザルトコードに対して定数 OVERRIDE_LAYOUT を追加します。OVERRIDE_BUFFER_PTR マクロを使用することで、ダウンロードバッファに対するポインタが利用できます。シンタックスは、この例のようにブロックがコンマによって区切られることを除いて、ファイル内（10 進のブロックカウントに続いて 16 進のブロックサイズ）と同様です。

```
uint32_t FlashInit(void *base_of_flash, uint32_t image_size)
{
strcpy(OVERRIDE_BUFFER_PTR, "2 0x100,7 0x200,7 0x1000");
return RESULT_OK | OVERRIDE_LAYOUT;
}
```

ギャップを指定するには、0 のブロックカウントを使用します。例えば、「0 0x1000」は 0x1000 バイトのギャップを指定します。

オーバーライドの結合

以上のオーバーライドはすべて、結合できます。この例ではすべてのオーバーライドが使用されています。

```
uint32_t FlashInit(void *base_of_flash, uint32_t image_size)
{
strcpy(LAYOUT_OVERRIDE_BUFFER, "2 0x100,7 0x200,7 0x1000");
SET_PAGESIZE_OVERRIDE(128);   // New page size
SET_BUFSIZE_OVERRIDE(0x1000); // New buffer size
return RESULT_OK | OVERRIDE_LAYOUT
| OVERRIDE_PAGESIZE | OVERRIDE_BUFSIZE;
};
```

フラッシュローダ自身のオーバーライド

フラッシュメモリデバイスがフラッシュローダの性能に不適合であることをフラッシュローダが検知する場合、一般的に、もっとも深刻なオーバーライドは、フラッシュローダが誤って起動することです。これは、通常、構成ミスの結果として発生するもので、ほとんどのフラッシュローダはこれをチェックすることもできません。しかし、フラッシュローダが実行時フラッシュメモリデバイスを検出できる場合、デバイスのレポートを C-SPY に送出して、C-SPY に別のフラッシュローダを起動させるよう指示することが可能です。これは、以下の例のように、バッファにデバイス識別子を入れて、特殊リザルトコード RESULT_OVERRIDE_DEVICE を戻すことで実施されます。

```
uint32_t FlashInit(void *base_of_flash, uint32_t image_size)
{
if ('unexpected flash device was found')
  {
strcpy(OVERRIDE_BUFFER_PTR, "P8_16c");
return RESULT_OVERRIDE_DEVICE;
  }
}
```

フラッシュローダ交換は、フラッシュメモリデバイス識別子として間接的に指定されることに注意してください。この識別子は C-SPY によって読まれ、その際、別のフラッシュローダを特定するためにテーブルルックアップ内のキーとして使用されます。テーブルは以下のように構築されます。

- C-SPY は、$TOOLKIT_DIR\config\flashloader ディレクトリ（および全サブディレクトリ）内で、ファイル名拡張子 flashdict を持つ全ファイルを発見します。
- このようなファイルの各々は、テーブルの各部の構築に寄与します。

テーブルは以下のようになります。

```
<?xml version="1.0" encoding="iso-8859-1"?>
<loaders>
```

```
<loader>
<key>P8_16c</key>
<path>$TOOLKIT_DIR$\config\flashloader\P8\f_p8_16c.flash</path>
</loader>
<loader>
<key>P8_16d</key>
<path>$TOOLKIT_DIR$\config\flashloader\P8\f_p8_16d.flash</path>
</loader>
</loaders>
```

キーがテーブル内のどこかで発見される場合、新規に指定されたフラッシュメモリ構成ファイルが替わりに使用されます。

Flags（フラグ）

FlashInit に対する flags パラメータは、オプションのフラグビットを指定します。現在、一個のフラグ値のみ定義されます。

FLAG_ERASE_ONLY	このビットが設定されれば（表現式（flags & FLAG_EASE_ONLY）がゼロでないとき）、フラッシュメモリ全体を消去するという唯一の目的によりフラッシュローダが呼び出されていることになります。フラッシュデバイスがワンステップグローバル消去ファンクションをサポートしている場合、フラッシュデバイスは、FlashInit から直接呼び出すことができ、これにより定数 RESULT_ERASE_DONE が戻されます。そうでない場合、C-SPY は動作を続け、各ブロックに対して FlashErase ファンクションを呼び出します。

FlashChecksum（フラッシュチェックサム）

これはオプションの関数です。ダウンロードしたフラッシュメモリ内容のチェックサム検証を有効にしたい場合、これをインプリメントする必要がありますが、以下のようにフレームワーク（Crc16）のヘルパーファンクションによってインプリメントすることができます。

```
OPTIONAL_CHECKSUM
uint32_t FlashChecksum(void const *begin, uint32_t count)
{
return Crc16((uint8_t const *)begin, count);
}
```

OPTIONAL_CHECKSUM マクロに注意してください。このマクロは、このようなオプションのファンクションや、そのフレームワークのラッパーが両方ともフラッシュローダのリンクに含まれていることを確認するために必要です。

FlashSignoff（フラッシュサインオフ）

これはオプションの関数です。フラッシュローディングが終了した後に何らかのクリーンアップを実施する必要がある場合、これをインプリメントできます。ファンクションは、FlashWrite への最後のコールが実施された後（または FlashChecksum の後（存在する場合））に呼び出されます。

```
OPTIONAL_SIGNOFF
uint32_t FlashSignoff()
{
return RESULT_OK;
}
```

OPTIONAL_SIGNOFF マクロに注意してください。このマクロは、このようなオプションのファンクションや、そのフレームワークのラッパーが両方ともフラッシュローダのリンクに含まれていることを確認するために必要です。

デバッギング

フラッシュローダは main ファンクションを持っているスタンドアロンプログラムではないため、デバギングはあまり簡単ではありません。

フラッシュローダを開発している間は、適切に準備あるいは生成されたデータやパラメータを備えた FlashInit、FlashWrite および FlashErase を単純に呼び出す main ファンクションを含んでいる非常に単純なテストハーネスを最初に作成するのが最善かもしれません。このプログラムは、RAM アドレスへリンクする必要があり、その後に、基本フラッシュプログラミングコードが正しくなるまで、通常アプリケーションとしてデバッグすることができます。

次に、フラッシュローダが実際のフラッシュローディングプロセスで使用される際、フラッシュローディングプロセスをある程度詳細に記述するログファイルからなんらかの助けを得ることでしょう。フラッシュローダを使用するデバッグセッションを開始すると、flash0.trace という名前のログファイルがプロジェクトディレクトリ（$PROJ_DIR$）に生成され、ディレクトリにはアクティブなプロジェクトファイル（ewp）が残ります。このファイルは、その名前のファイルがすでにそのディレクトリに存在している場合のみ生成されます。トレース出力を有効にするには、flash0.trace という名前の空ファイルを作成します。フラッシュローディングによるデバッグセッションが開始されるたびに、ファイルが再び除去されるまで、トレース出力が生成されます。

複数のフラッシュローディングパスが存在する場合、複数のトレースファイルが生成されますが（flash0.trace、flash1.trace、その他 ）、トレースを有効にするには、flash0.trace を作成するだけで大丈夫です。

次は、ログファイルの一例です。

```
File generated Wed Dec 09 14:53:55 2009

Pass 1 of 1
Starting fragment-style flashloader pass.
FlashInitEntry is at 0x2000012D
FlashWriteEntry is at 0x20000135
FlashEraseWriteEntry is at 0x2000013D
FlashBreak is at 0x20000128
FlashBufferStart is at 0x20000200
FlashBufferEnd is at 0x2000BD94
FlashChecksumEntry not found
FlashSignoffEntry not found
page size is 2 (0x2)
filler is 0xff
buffer size is 48020 (0xbb94)
SimpleCode records (after offset):
  Record 0: @ 0x8000000 [53284 (0xd024) bytes] 0x8000000 - 0x800d023 [0
4 0]
Base of flash at 0x8000000
->init       : base @ 0x8000000, image size d024
      Args: (argc = 1)
          C:\sample\blabla.out
  timing(init): 0.0156 (CPU) 0.0500 (elapsed)
Transaction list:
  Transaction @ 0x8000000 (0xb800 bytes) 23 packet(s).
    Will erase 23 block(s):
      0: 0x8000000 (0x800 bytes)
      1: 0x8000800 (0x800 bytes)
      2: 0x8001000 (0x800 bytes)
```

```
      3: 0x8001800 (0x800 bytes)
      4: 0x8002000 (0x800 bytes)
      5: 0x8002800 (0x800 bytes)
      6: 0x8003000 (0x800 bytes)
      7: 0x8003800 (0x800 bytes)
      8: 0x8004000 (0x800 bytes)
      9: 0x8004800 (0x800 bytes)
      10: 0x8005000 (0x800 bytes)
      11: 0x8005800 (0x800 bytes)
      12: 0x8006000 (0x800 bytes)
      13: 0x8006800 (0x800 bytes)
      14: 0x8007000 (0x800 bytes)
      15: 0x8007800 (0x800 bytes)
      16: 0x8008000 (0x800 bytes)
      17: 0x8008800 (0x800 bytes)
      18: 0x8009000 (0x800 bytes)
      19: 0x8009800 (0x800 bytes)
      20: 0x800a000 (0x800 bytes)
      21: 0x800a800 (0x800 bytes)
      22: 0x800b000 (0x800 bytes)
  Transaction @ 0x800b800 (0x1824 bytes) 4 packet(s).
    Will erase 4 block(s):
      0: 0x800b800 (0x800 bytes)
      1: 0x800c000 (0x800 bytes)
      2: 0x800c800 (0x800 bytes)
      3: 0x800d000 (0x800 bytes)
->multi_erase: 23 blocks (0xb8 bytes in buffer) [0 0 0]
  timing(erase): 0.1716 (CPU) 0.5460 (elapsed)
->write      : @ 0x8000000 (0xb800 bytes, offset 0x0 into block @
0x8000000) [0 4 0]
  timing(write): 0.0000 (CPU) 1.3960 (elapsed)
->multi_erase: 4 blocks (0x20 bytes in buffer) [0 b8 0]
  timing(erase): 0.0000 (CPU) 0.1310 (elapsed)
->write      : @ 0x800b800 (0x1824 bytes, offset 0x0 into block @
0x800b800) [1b 68 1b]
  timing(write): 0.0000 (CPU) 0.2200 (elapsed)
Duration:   1.28 (CPU)   4.34 (elapsed)
  of which on target: 0.1872 (CPU) 2.3430 (elapsed)
Flash loading pass finished
```

最初に、ファイルは、フラッシュローダが持ついくつかの主要ファンクションのアドレス、およびフラッシュメモリやフラッシュローダの基本的なプロパティのいくつかをリスト化します。次に、ダウンロードされるイメージのデータレコードがリスト化されます。中心となる部分は、書き込みおよび消去操作シーケンスで、各々は開始アドレス、サイズ、さらには行の最後に、操作に対するデータの最初のバイト三個を含んでいます。リストファイルの最後に、チェックサム操作が任意にリスト化されます。