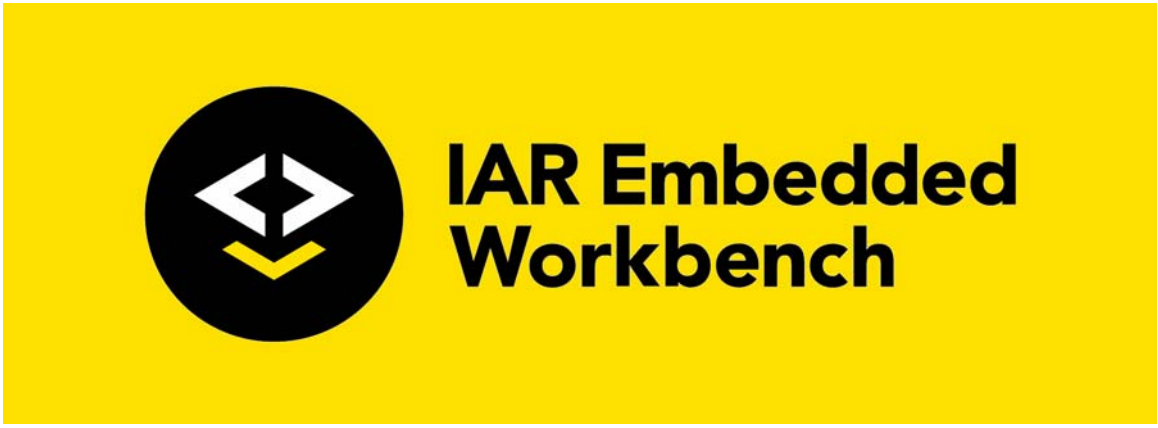

移行ガイド

Advanced RISC Machines Ltd

ARM コア

MARM-4-J

IAR SYSTEMS

改版情報
第 4 版 : 2017 年 3 月

部品番号 : MARM-4-J

本ガイドは、ARM 用 IAR Embedded Workbench® のバージョン 8.1x に適用する。

内部参照 : BB1、Mym8.0、IMAE。

バージョン 6.x または 7.x から 8.x への移行

この章では、ARM 用 IAR Embedded Workbench の旧バージョン 6.x と 7.x から、新しいバージョン 8.x にアプリケーションコードとプロジェクトを移植するためのヒントを提供します。

6.x より前のバージョンから移行する場合は、先に本ガイドの以前の移行の章に目を通しておく必要があります。

バージョン 6.x から 7.x への移行には問題はありません。

移行についての考慮事項

古いプロジェクトを移行するために、以下のトピックを考慮してください。

- IAR Embedded Workbench IDE
- C/C++ 言語の変更
- 変更されたオプション
- ライブラリ構造の変更

このリストにあるすべての項目がユーザのプロジェクトに適切なわけではありません。それぞれのケースにどのアクションが必要か、慎重に考えてください。

バージョン 6.x または 7.x 用に記述されたコードでは、バージョン 8.x ではワーニングやエラーを出力することがあります。

IAR Embedded Workbench IDE

新しいバージョンの IAR Embedded Workbench IDE にアップグレードする場合、このセクションに記載された事項を考慮する必要があります。

インストール先ディレクトリ

新しいバージョンの IAR Embedded Workbench IDE をインストールするときは、旧バージョンと同じインストール先ディレクトリにはインストールできません。

旧バージョンのデフォルトのインストールパス：

`c:¥Program Files¥IAR Systems¥Embedded Workbench 6.`*n*`¥`

または

`c:¥Program Files¥IAR Systems¥Embedded Workbench 7.`*n*`¥`

新バージョンのデフォルトのインストールパス：

`c:¥Program Files¥IAR Systems¥Embedded Workbench 8.`*n*`¥`

IAR Embedded Workbench のバージョン番号の違いに注意してください。

C/C++ 言語の変更

C

バージョン 8.x では、コンパイラはデフォルトで C11 規格 (ISO/IEC 9899:2012) に準拠しています（これを本ガイドでは以降 *C 規格*と呼びます）。

非 AEABI モードのシステム ヘッダーに使用している、C 言語の古いバージョンを使用してコンパイルされた古いオブジェクトファイルは、再コンパイルする必要があります。

C++

バージョン 8.x では、コンパイラは C++14 規格 (ISO/IEC 14882:2014(E)) を使用し、古い C++03 規格の代わりにこのガイドの *C++ 規格*として参照します。

C++03 規格を使用してコンパイルされた古いオブジェクトファイルは、再コンパイルする必要があります。

C++03 規格で記述されたソースファイルは、C++14 言語でほぼ意図したとおりにコンパイルされます。C++14 規格には構文の変更があり、C++03 のソースファイルをコンパイルできない、または実行時に失敗することがあります。

例：

- `X<1>>2>>x;` は現在無効です
- `auto` は、型推論およびスタックを維持しないことを意味します

Embedded C++

Embedded C++ (EC++) と拡張された Embedded C++ (EEC++) はサポートされなくなりました。IAR Embedded Workbench プロジェクトの更新では、自動的に **EC++** または **EEC++** 言語のオプションを **C++** に変更します。コマンドライ

ンでは、オプション --ec++ と --eec++ を --c++ --no_exceptions --no_rtti に変更する必要があります。

EC++ または EEC++ 言語を使用してコンパイルされた古いオブジェクトファイルは、再コンパイルする必要があります。

EC++ または EEC++ 言語で記述されたソースファイルは、C++ 規格にポートされる必要があります。ライブラリのシンボルは、現在 namespace std に存在します。これを解決するには、ライブラリシンボルが接頭辞リファレンス std:: を参照するようにするか、using namespace std; を、C++ システムヘッダーの最後に含まれているものの後に挿入します。4 ページの *C++* を参照してください。

変更されたオプション

マルチバイト文字サポート

コンパイラオプション --enable_multibytes は削除されています。オプションは次のことを決定します。

- コンパイラは、文字列リテラル、文字定数、およびソースファイルのコメントに、Raw エンコード、またはシステムのデフォルトのロケール エンコードのどちらを使用するのか。
- printf と scanf のフォーマッタ派生型の自動選択は、マルチバイトおよび wchar_t フォーマッタをサポートするかどうか。

考慮する該当のオプションを次に示します。

- なし、Raw エンコードをソースファイルに使用する必要がある場合。
- --source_encoding locale、ソースファイルにシステムのデフォルトのロケールを使用する必要がある場合。

お使いの IAR Embedded Workbench プジェクトを更新する場合、--source_encoding オプションは自動的に設定されます。

コンパイラオプション --printf_multibytes および --scanf_multibytes を使用して、printf および scanf フォーマッタの自動選択がマルチバイトをサポートするかどうかを決定します。

プリプロセッサ出力

コンパイラオプション --preprocess には現在異なるパラメータのセットがあります。

c	コメントを含む
n	プリプロセスのみ
s	#line ディレクティブを無効化

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［プリプロセッサ］>［ファイルへのプリプロセッサ出力］

［言語］オプション

--ec++

コンパイラオプション --ec++ は削除されています。--no_exceptions の --c++ および --no_rtti を代わりに使用します。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［C++］

--eec++

コンパイラオプション --eec++ は削除されています。--no_exceptions の --c++ および --no_rtti を代わりに使用します。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［C++］

ライブラリ構造の変更

これらの変更は、ライブラリソースコードの構造で行われています。

- システムのヘッダーyfuns.h は現在 LowLevelIOInterface.h という名前です。
- 定義 _STD_BEGIN および _STD_END は削除されています。ソースコードから削除する、またはそれらをなしと定義します。

バージョン 5.x 6.x への移行

この章では、ARM 用 IAR Embedded Workbench の旧バージョン 5.x から、新しいバージョン 6.x にアプリケーションコードとプロジェクトを移植するためのヒントを提供します。

5.x より前のバージョンから移行する場合は、先に本ガイドの以前の移行の章に目を通しておく必要があります。

移行についての考慮事項

古いプロジェクトを移行するために、以下のトピックを考慮してください。

- IAR Embedded Workbench IDE
- C/C++ 言語の変更
- ランタイムライブラリの変更

このリストにあるすべての項目がユーザのプロジェクトに適切なわけではありません。それぞれのケースにどのアクションが必要か、慎重に考えてください。

バージョン 5.x 用に記述されたコードでは、バージョン 6.x ではワーニングやエラーを出力することがあります。

ユーザードキュメンテーションの変更

『*IAR Embedded Workbench® IDE User Guide for ARM®*』は、以下に置き換えられました。

- 『*C-SPY® Debugging Guide for ARM®*』によって、デバッグに関するすべての情報が置き換えられます
- 『*ARM® 用 IDE プロジェクト管理およびビルド*』によって、IDE におけるプロジェクト管理とビルドに関するすべての情報が置き換えられます

IAR Embedded Workbench IDE

新しいバージョンの IAR Embedded Workbench IDE にアップグレードする場合、このセクションに記載された事項を考慮する必要があります。

インストール先ディレクトリ

新しいバージョンの IAR Embedded Workbench IDE をインストールするときは、旧バージョンと同じインストール先ディレクトリにはインストールできません。

旧バージョンのデフォルトのインストールパス：

`c:¥Program Files¥IAR Systems¥Embedded Workbench 5.`*n*`¥`

新バージョンのデフォルトのインストールパス：

`c:¥Program Files¥IAR Systems¥Embedded Workbench 6.`*n*`¥`

IAR Embedded Workbench のバージョン番号の違いに注意してください。

［オプション］ダイアログボックスのプロジェクト設定

［**オプション**］ダイアログボックス（［**プロジェクト**］メニューから表示）が変更になりました。次の表は最も重要な変更点の一覧です。

カテゴリ > ページ	変更点
C/C++ コンパイラ > 言語	8 ページの *C/C++ 言語の変更*を参照してください。

表 1: ［プロジェクト］オプションダイアログボックスの変更の概要

プロジェクトファイル

オプションダイアログボックスのページの一部は変更になっていますが、古いプロジェクトファイルを新しいバージョンの IAR Embedded Workbench IDE で使用できます。

古いプロジェクトを変換すると、新しいオプションの **C++ インライン動作**が有効になります。ソースコードを C 規格 (C99) に準拠させるには、必ずこのオプションを無効にしてください。詳しくは、『*ARM 用 IAR C/C++ 開発ガイド*』の **--use_c++_inlines** を参照してください。

C/C++ 言語の変更

バージョン 6.x では、コンパイラはデフォルトで C99 規格（ISO/IEC 9899:1999 technical corrigendum No.3 を含む）に準拠しています（これを本ガイドでは以降 *C 規格*と呼びます）。オプションで、コンパイラを C89 規格（ISO 9899:1990 すべての technical corrigenda および addenda も含む）に準拠されることもできます。C89 モードでは、C99 言語の機能や C99 のライブラリシンボルを使用することはできません。

バージョン 5.x では、コンパイラはデフォルトで C89 規格に準拠しています。オプションで C99 の一部の機能を使用できます。

バージョン 6.x に移行するには、以下を考慮する必要があります。

- 言語サポートのオプション
- 言語の適合オプション
- ソースコード内の古い C89 機能

言語サポートのオプション

次の表は、言語サポートを有効にするためのオプションの違いの一覧です。

言語機能	バージョン 6.x	バージョン 5.x
C89*	`--c89`†	デフォルトでサポート。
C99* （C 規格）	デフォルトでサポート	`-e` の使用時に一部の機能を使用できます。

表 2: 言語機能の有効化

* C89 と C99 にはそれぞれ、軽度の例外がいくつかあります。詳細については、コンパイラのドキュメントを参照してください。

† `--c89` は C99 のライブラリシンボルと C99 の言語機能を無効にします。

注 : C99 モードでは、可変長配列 (VLA) はデフォルトでは許可されていません。こうしたサポートを有効にするには、IDE でコマンドラインオプション `--vla` または **VLA の許可**を使用してください。

言語の適合オプション

C/C++ 言語の適合オプションは、新旧のバージョンで異なります。次の表は、この違いを示します。

バージョン 6.x IDE と コマンドラインの オプション比較	バージョン 5.x IDE と コマンドラインの オプション比較	説明
標準（IAR 拡張あり） `-e`	**IAR 拡張を許可する** `-e`	IAR 拡張および C 規格への IAR による緩和を受け入れます。
標準 デフォルトでコマンドラインによりサポート	**ISO/ANSI に緩く準拠** デフォルトでコマンドラインによりサポート	C 規格への IAR による緩和を受け入れます。
厳密 `--strict`	**ISO/ANSI に厳密に準拠** `--strict_ansi`	規格に厳密に準拠します。

表 3: 言語の適合オプション

Embedded C++ および C++ 規格

C++ 規格は、バージョン 6.x でサポートされています。C++ 規格の使用について詳しくは、『*ARM 用 IAR C/C++ 開発ガイド*』を参照してください。旧式の Embedded C++ プロジェクトを C++ 規格のプロジェクトに自動的に変換する方法はありません。また、EC++ ライブラリは C++ ライブラリとは互換性がありません。Embedded C++ の実装が変更になっています。

C99 インライン

バージョン 5.x では、インラインバージョンの関数は、どこで定義されていても同じコードを持つことになっています。リンク時には、これらのすべてが非インラインバージョンとして使用できます。C99 では、インライン関数は定義される場所によって異なるコードを持つ可能性がありましたが、リンク時には非インラインバージョンが 1 つだけになります（`extern inline` により宣言されたもの）。次に例を示します。

```
static int x;
inline void f()
{
  //static int y; // 静的を参照できない
  //x;            // 静的を参照できない
}

// この f は使用する非インラインバージョンとして宣言
extern inline void f();
```

ソースコード内の古い C89 機能

バージョン 5.x でコンパイラにより受け付けられる C89 の機能はいくつかありますが、C99 モードでコンパイルする際には、これらはバージョン 6.x では受け付けられません。ワーニングやエラーが出力されます。これらの診断メッセージを省略するには、ソースコードを C89 モードでコンパイルするか、記述し直す必要があります。

これらの C89 機能は、C99 モードでコンパイルするときにはバージョン 6.x では受け付けられません。

- 暗黙的な `int` 変数

  ```
  static k; /* k は暗黙的 int です */
  ```

- 暗黙的な `int` パラメータ

  ```
  myFunction(i,j)
  {
    /* i と j は暗黙的な int です */
  }
  ```

- 暗黙的な int のリターン

```
myFunction()
{
  /* 暗黙的な int 0 が返されます */
}
```

- 暗黙的なリターン

```
myFunction()
{
  /* 0 が返される */
}
```

- 暗黙的なリターン

```
myFunction()
{
  return; /* 0 が返される */
}
```

ランタイムライブラリの変更

バージョン 6.x では、コンパイラとアセンブラは事前に決定されたディレクトリ（コンパイラ / アセンブラの実行可能ファイルに相対）でシステムヘッダファイルを自動的に検索します。バージョン 5.x では、インクルードファイルの検索パスを明示的に指定する必要があります。

バージョン 6.x では、このために以下のコンパイラオプションが用意されています。

`--dlib_config` *token*
: DLIB システムヘッダファイルを使用します。このオプションでは、使用するランタイムライブラリ構成も指定できます。バージョン 5.x では、このオプションでランタイムライブラリ構成ファイルを指定できますが、バージョン 6.x では、このオプションはトークンも受け入れます。

`--no_system_include`
: システムヘッダファイルの自動検索を無効化します。バージョン 5.x のときと同じように、インクルードファイルの検索パスを明示的に指定する必要があります。このオプションは、アプリケーションプロジェクトのビルド用に実績のあるスクリプトファイルがあって、それを新しいインクルードファイルシステムにすぐに適用したくない場合に便利です。

`--system_include_dir`	コンパイラがシステムヘッダファイルを検索可能なインクルードディレクトリを明示的に指定します。

以下の対応するアセンブラオプションが用意されています。

`-g`	システムヘッダファイルの自動検索を無効化します。バージョン 5.x のときと同じように、インクルードファイルの検索パスを明示的に指定する必要があります。このオプションは、アプリケーションプロジェクトのビルド用に実績のあるスクリプトファイルがあって、それを新しいインクルードファイルシステムにすぐに適用したくない場合に便利です。
`--system_include_dir`	`inc` ディレクトリを明示的に指定します。このディレクトリで、アセンブラがシステムヘッダファイルを検索できます。

これらのオプションについて詳しくは、『*ARM® IAR C/C++ Compiler Reference Guide*』および『*ARM® IAR Assembler Reference Guide*』を参照してください。

バージョン 4.x から 5.x への移行

このガイドでは、ARM IAR Embedded Workbench® バージョン 4.x と ARM IAR Embedded Workbench バージョン 5.x の大きな違いと、移行に関する考慮事項について説明します。主に、以下の方法を説明します。

- 既存のアプリケーションソースコードを問題なくコンパイル、リンクする。
- ランタイムでの動作の潜在的な変化を特定する。

このガイドは、アプリケーションのソースコードおよび他のプロジェクトファイルを、新しいバージョン 5.x に移植するための情報を提供します。

ARM IAR Embedded Workbench バージョン 3.x から移行する場合、まず *ARM® IAR Embedded Workbench バージョン 4.x への移行*の章に目を通してください。

移行プロセス

バージョン 4.x と 5.x との概念上の違いは、IAR ビルドツールで使用する内部オブジェクトフォーマットが変わった点です。バージョン 4.x では、IAR システムズのフォーマット UBROF が使用され、バージョン 5.x では業界標準のフォーマット、デバッグ情報用の DWARF (ELF/DWARF) を含む *Executable and Linking Format* が使用されます。これは、IAR XLINK リンカに置き換わる完全に新しいリンカ（IAR ILINK リンカ）ということを意味します。

オブジェクトフォーマットは、ELF/DWARF もサポートする他のベンダ製のツールと互換性を持つように変更されました。

これらの違いによって、アプリケーションのソースコードと他の関連プロジェクトファイルを修正せざるを得なくなります。つまり、バージョン 4.x から 5.x に移行するには、以下の変更を考慮する必要があります。

- コンパイラと C ソースコード
- アセンブラおよびアセンブラソースコード
- リンカとリンカの設定
- ランタイム環境とオブジェクトファイル
- IAR Embedded Workbench IDE のプロジェクトファイルおよびプロジェクト設定
- デバッガ

古いプロジェクトを移行するには、記載された移行プロセスに従います。説明されている移行プロセスのすべての手順が該当するとは限らないため、注意してください。それぞれのケースにどのアクションが必要か、慎重に考えてください。

注 : Version 5.x:

- ライブラリを管理するための IAR システムズのアプリケーション XAR と XLIB は、IAR アーカイブビルダ (`iarchive`) および IAR の製品のインストールとともに提供される GNU ユーティリティのセットに置き換わりました。
- ユーザドキュメンテーションの構成が変わりました。以前は、コンパイラとリンカは 2 つの別のガイドになっていました。現在はコンパイラとリンカは両方とも、『*ARM 用 IAR C/C++ 開発ガイド*』で解説されています。

コンパイラと C ソースコード

C/C++ ARM IAR C/C++ コンパイラバージョン 4.x 用に記述されたソースコードは、ARM IAR C/C++ コンパイラの新バージョン 5.x でも使用できます。ただし、新しいコンパイラを使用して既存のソースコードをコンパイルする前に、以下の点について確認する必要があります。

C/C++ ソースコードのファイルで、以下の変更に注意してください。

- 絶対アドレスに配置された定数ではイニシャライザは使用できなくなりました。すなわち、以下のタイプの構造は使用できません。

  ```
  int const a @ 10 = 20;
  ```

- `#pragma vector` ディレクティブは、使用できなくなりました。割込みベクタの名前はあらかじめ決められています。たとえば、`cstartup.o` で定義された `IRQ_Handler` というようにです。
- `#pragma swi_number` ディレクティブと `__swi` キーワードは現在、関数の宣言でのみ使用できます。関数の定義では使用できません。

- 組込み関数 __enable_interrupt と __disable_interrupt は、組込み関数としては使用できなくなりました。ただし、ライブラリ関数としては利用できます。つまり、ソースコードのレベルで下位互換性があります。
- __monitor キーワードは、現在は使用できません。このキーワードは、次のコードシーケンスに置き換えることができます。

```
void function_that_was_declared_monitor_in_ewarm_4xx()
{
  __istate_t isate = __get_interrupt_state();
  __disable_interrupt();
  ...
  __set_interrupt_state(istate);
}
```

- #pragma optimize ディレクティブの動作が変わりました。以下のパラメータはは認識されますが、バージョン 5.x では無視されます。s（速度）、z（サイズ）、2（なし）、3（低）、6（中）、9（高）。バージョン 5.x では、これらはパラメータ speed、size、および balanced（速度とサイズのバランスを調整）に置き換わりました。これらをパラメータ high と組み合わせることができます。
- セグメント演算子 __segment_size は使用できなくなりました。この演算子を置き換えるには、以下のタイプの構造を使用できます。

```
size = __section_end("xxx") - __section_begin("xxx");
```

- ビットフィールドのデフォルトのレイアウトが、disjoint_types から joint_types に変わりました。ただし、移行のために、外部インタフェースの一部であるビットフィールドには注意する必要があります。この場合は、ビットフィールドについて『*ARM 用 IAR C/C++ 開発ガイド*』を参照してください。

  リトルエンディアンモードでは、すべてのビットフィールドが同じ基本型を持つ構造のレイアウトは以前と同じです。

2 セグメントの代わりに、コンパイラはコードとデータをセクション内に配置するようになりました。定義済みのセグメント名を明示的にソースコードで使用していない限り、この内部変更によって、使用している C/C++ ソースコードを変更する必要はありません。変更する必要がある場合は、必ず新しいセクション名を使用してください（28 ページの *セグメントとセクションの比較*を参照）。

初期化済みのセグメントの扱いも変わりました（28 ページの *初期化用のセグメント*を参照）。

3 コンパイラのオプションに関する変更がいくつかあります。削除されたオプションや変更になったオプションのほか、新しいオプションもいくつかあります。変更の一覧については、24 ページの *ツールオプション*を参照してください。

4 ファイル名拡張子に関する変更について詳しくは、31 ページの *ファイル名拡張子*を参照してください。

5 I/O 定義ヘッダファイルのディレクトリ構造が変わりました。バージョン 5. x では、これらのファイルはデバイス固有のサブフォルダに配置されます。これによって検索パスが影響を受けるため、それに応じてヘッダファイルを変更する必要があります。次に例を示します。

```
#incude <ioml671000.h>
```

は次のように変更してください。

```
#incude <oki/ioml671000.h>
```

バージョン 5.x の機能について詳しくは、『*ARM 用 IAR C/C++ 開発ガイド*』を参照してください。

アセンブラおよびアセンブラソースコード

アセンブラ実行可能ファイルの名前が、`aarm` から `iasmarm` に変わりました。

アセンブラソースコードで、以下の点を考慮してください。

1 モジュール

バージョン 5.x では、アセンブラやコンパイラはプログラムとライブラリ *モジュール*の区別ができません。モジュールをライブラリモジュールとして扱う（つまり、条件付きでリンクする）場合は、モジュールをライブラリに配置する必要があります。

これは、既存のアセンブラソースコードで `LIBRARY` または `MODULE` ディレクティブを使用したことがある場合、何の効果もなくなるということです。

バージョン 4.x では、各ファイルに 1 つまたは複数のアセンブラモジュールを定義できました。バージョン 5.x では、ファイルごとに 1 つのモジュールしか定義できません。つまり、それに応じてファイルを再構成しなければなりません。

モジュール構造のプログラミングおよびモジュールディレクティブの新しい構文については、『*ARM® IAR Assembler Reference Guide*』を参照してください。

2 セグメントとセクションの比較

セグメントの概念は、セクションという概念になりました。すなわち :

- セグメント上で動作するアセンブラディレクティブは、削除またはセクション上で動作する新しいディレクティブに置き換わりました。つまり、それに応じてアセンブラソースコードを修正する必要があります。詳しく

は、『*ARM® IAR Assembler Reference Guide*』のセクション制御のディレクティブの説明を参照してください。

- アセンブラのソースコードで、バージョン 4.x に固有の事前定義されたセグメントを使用したことがある場合、古いセグメント名をすべて新しいセクション名に置き換える必要があります。詳細については、28 ページの *セグメントとセクションの比較*を参照してください。

3 式

バージョン 5.x では、1 つの式で 2 つのシンボルを使用することはできません。または、式がアセンブリ時に解決できない限り、複雑な式を使用することもできません。そのような式は記述し直す必要があります。そうしなければ、アセンブラはエラーを出力します。リンク時に解決できない式を以下の例で説明します。

```
        public glob_var
        extern ext_var
var1    DC32    glob_var + ext_var   ; 失敗します
var2    DC32    glob_var * 5 + 3     ; 失敗します
var3    DC32    glob_var + 3         ; OK
```

4 アセンブラディレクティブ

一部のアセンブラディレクティブは削除され、一部は新しい構文を使用するか、あるいは他の変更が加えられています。バージョン 4.x で 5.x のときと異なるアセンブラディレクティブの一覧は、30 ページの *アセンブラディレクティブ*を参照してください。

これらのディレクティブのいずれかをアセンブラソースコードで使用したことがある場合、これらの構造を記述し直す必要があります。

これらのディレクティブの詳細については、『*ARM® IAR Assembler Reference Guide*』を参照してください。

5 定義済シンボル

定義済シンボル `__ASMARM__` は、シンボル `__IASMARM__` に置き換えられました。

6 呼び出し規約

コンパイラで使用する呼出し規約が変わりました。バージョン 4.x では、スタックはデフォルトで 4 にアラインメントされますが、Advanced RISC Machines Ltd Arm/Thumb のプロシージャ呼出し基準 (ATPCS) に従って、スタックを 8 にアラインメントすることが可能です。バージョン 5.x では、コンパイラは ARM アーキテクチャのプロシージャ呼出し基準 (AAPCS) に準拠します。つまり、スタックは 8 バイトにアラインメントできるようになりました。

アセンブラ関数を呼び出す C 関数や、その逆の関数がある場合、新しい呼出し規約に従ってアセンブラのルーチンを記述し直す必要があります。

こうしたプロシージャの呼出し基準やそれらの違いについて詳しくは、『*ARM 用 IAR C/C++ 開発ガイド*』のほか、Web サイト `www.arm.com` も参照してください。

7 C-SPY の［コールスタック］ウィンドウのバックトレース情報

C-SPY の［コールスタック］ウィンドウのバックトレース情報に対するコンパイラのリソース名が標準化され、`CfiCommon.h` に定義されています。つまり、独自のリソース名は定義できなくなりました。`CFI` アセンブラディレクティブを使用して*名前オプション*を定義したことがある場合、これに標準化されたリソース名のサブセットが含まれている必要があります。標準化されたリソース名の一覧については、『*ARM 用 IAR C/C++ 開発ガイド*』を参照してください。

8 ファイル名拡張子に関する変更について詳しくは、31 ページの *ファイル名拡張子*を参照してください。

9 環境変数 `ASMARM` と `Aarm_INC` は、それぞれ `IASMarm` と `IASMarm_INC` に変わりました。

リンカとリンカの設定

IAR XLINK リンカは、IAR ILINK リンカに置き換えられました。

XLINK と ILINK の比較

XLINK と ILINK は両方とも、1 つまたは複数の再配置可能なオブジェクトファイルを、1 つまたは複数のオブジェクトライブラリの選択した部分と組み合わせて、実行可能イメージを生成します。XLINK は、IAR システムズのツールにより生成された UBROF フォーマットのオブジェクトファイルのみ受け入れ、出力フォーマット UBROF またはサポートされているその他の出力フォーマットで出力を生成します。ILINK は ELF フォーマットのオブジェクトファイルを受け入れ、実行可能イメージを ELF フォーマットで生成します。

バージョン 4.x では、コンパイラはコードとデータを UBROF *セグメント*に配置します。XLINK は、これを*リンカコマンドファイル*で指定されたディレクティブに従って、メモリ内に配置します。このファイルはコマンドラインの拡張であり、その中で任意の XLINK コマンドラインオプションを指定できます。バージョンでは、コンパイラはコードとデータを ELF *セクション*に配置します。ILINK は、*ILINK 設定ファイル*で指定した*設定成*に従って、これらのセクションを配置します。このファイルではアプリケーションの初期化

フェーズの自動処理もサポートしています。すなわち、イニシャライザのコピーや、場合によっては解凍も行って、グローバル変数領域とコード領域のイニシャライズを行います。ただし、ファイルにはコマンドラインを一切含めることはできません。これらは、コマンドラインで指定する必要があります。

XLINK から ILINK への移行

1. 新しい IAR ILINK リンカはターゲット固有であり、IAR XLINK リンカに代わるものです。実行可能ファイルの名前は `xlink` から `ilinkarm` に変更されました。
2. リンカコマンドファイルを新しい ILINK 設定ファイルに移行するには、19 ページの *XLINK.XCL から ILINK.ICF への変換*の例を参照してください。
3. セグメントをセクションにマッピングする方法については、28 ページの *セグメントとセクションの比較*を参照してください。
4. ファイル名拡張子に関する変更について詳しくは、31 ページの *ファイル名拡張子*を参照してください。
5. ELF 出力を Intel-hex または Motorola S-records に変換する必要がある場合、付属の GNU ユーティリティを使用してください。

   ILINK の高度な機能について詳しくは、『*ARM 用 IAR C/C++ 開発ガイド*』のリンクについての章を参照してください。

XLINK.XCL から ILINK.ICF への変換

リンカコマンドファイル (XLINK) とリンカ設定ファイル (ILINK) は 2 つの異なる枠組みに基づいているため、リンカコマンドファイルの内容は自動的に変換されません。その代わりに、リンカ設定を手動で変換する必要があります。

IAR Embedded Workbench IDE を使用している場合、リンカ設定ファイルエディタを使用してリンカ設定を指定できます。

1. **[プロジェクト] > [オプション]** を選択して、**[リンカ]** カテゴリを選び、**[設定]** タブをクリックします。
2. リンカ設定ファイルエディタを開くには、**[デフォルトのオーバライド]** オプションを選択して **[編集]** ボタンをクリックします。
3. 表示されるダイアログボックスで、以下を定義します。
   - 割込みベクタテーブルの開始アドレス
   - RAM および ROM メモリ領域の開始 / 終了アドレス
   - スタックおよびヒープのサイズ

4 終わったら、**[保存]** ボタンをクリックします。初めてこれを行うときは、**[名前を付けて保存]** ダイアログボックスが表示されます。

注 : すべてのビルド構成について、専用のリンカ設定ファイルを明示的に選択する必要があります。

コマンドラインからプロジェクトをビルドする場合、リンカ設定ファイル generic.icf（arm¥config ディレクトリ内）あるいは examples ディレクトリのサンプルプロジェクトにある任意の設定ファイルを使用できます。これらの設定ファイルは、いずれもターゲットハードウェアおよびアプリケーション要件に適した設定ファイルを作成するためのテンプレートとして使用できます。

以下は、XLINK リンカコマンドファイル (xcl) および対応する ILINK リンカ設定ファイル (icf) の一例です。

```
-!=========================
-!xlink  xcl file
-!=========================
-carm

-DROMSTART=08000
-DROMEND=FFFFF

-Z(CODE)INTVEC=00-3F
-Z(CODE)ICODE,DIFUNCT=ROMSTART-ROMEND
-Z(CODE)SWITAB=ROMSTART-ROMEND
-Z(CODE)CODE=ROMSTART-ROMEND
-Z(CONST)CODE_ID=ROMSTART-ROMEND
-Z(CONST)INITTAB,DATA_ID,DATA_C=ROMSTART-ROMEND
-Z(CONST)CHECKSUM=ROMSTART-ROMEND

-DRAMSTART=100000
-DRAMEND=7FFFFF

-Z(DATA)DATA_I,DATA_Z,DATA_N=RAMSTART-RAMEND
-Z(DATA)CODE_I=RAMSTART-RAMEND
-QCODE_I=CODE_ID

-D_CSTACK_SIZE=2000
-D_IRQ_STACK_SIZE=100
-D_HEAP_SIZE=8000

-Z(DATA)CSTACK+_CSTACK_SIZE=RAMSTART-RAMEND
-Z(DATA)IRQ_STACK+_IRQ_STACK_SIZE,HEAP+_HEAP_SIZE=RAMSTART-RAMEND
```

以下は対応する icf ファイルです。

```
//=========================
//ilink icf ファイル
//-carm は移行に適していません。ilinkarm は
// ARM 専用であるためです。
//=========================
define memory mem with size = 4G;

define region ROM_region = mem:[from   0x8000 to  0xFFFFF];
define region RAM_region = mem:[from 0x100000 to 0x7FFFFF];

initialize by copy { rw };
do not initialize  { section .noinit };

define block CSTACK    with alignment = 8, size = 0x2000 { };
define block IRQ_STACK with alignment = 8, size =  0x100 { };
define block HEAP      with alignment = 8, size = 0x8000 { };

place at address mem:0x0 { ro section .intvec };
place in ROM_region      { ro };
place in RAM_region     { rw, block CSTACK, block IRQ_STACK,
                          block HEAP };
```

IAR Embedded Workbench IDE のプロジェクトファイルおよびプロジェクト設定

IAR Embedded Workbench IDE の新バージョンへのアップグレードには、手動による調整が少し必要です。

プロジェクトファイルの変換

IAR Embedded Workbench IDE を使用している場合、新しいバージョンの ARM IAR Embedded Workbench IDE を起動して古いワークスペースを開きます。バージョン 4.x で作成された古いプロジェクトを含むワークスペースを開くと、プロジェクトファイルをバージョン 5.x に変換するかどうかを確認するダイアログボックスが表示されます。**[OK]** をクリックすると、まず古いプロジェクトのバックアップが作成され、続いてプロジェクトが変換されます。

プロジェクトのオプションの移行

バージョン 4.x と 5.x では使用可能なツールオプションが異なるため、古いプロジェクトを変換した後はオプション設定を確認してください。

コマンドラインインタフェースを使用している場合、makefile ファイルを 24 ページの *ツールオプション*のマッピング表と比較し、それに従って makefile を変更するだけです。

IAR Embedded Workbench IDE を使用している場合は、両バージョンに共通のオプションはプロジェクトの変換時に自動的に変換されます。変更になったオプションは、デフォルト値に設定されます。

オプションを手動で確認するには、以下の手順に従います。

- **［コンパイラ］**カテゴリ

  ［コード］ページは新しいものですが、オプションは以前に**［一般］**カテゴリにありました。これらのオプションの設定は保持されます。

  ［最適化］ページは変更されました。`-s` および `-z` オプションに代わって、`-O` コンパイラオプションが導入されました（24 ページの *-s と -z および-O の比較*を参照）。

  ［出力］ページは変更になっています。独自のセグメント名を定義したことがある場合、これは自動的にはセクション名に変換されません。デフォルトのコードセクション名は `.text` です。セグメント名とセクション名について詳しくは、28 ページの *セグメントとセクションの比較*を参照してください。

- **［リンカ］**カテゴリ

  リンカオプションは自動的には変換されません。プロジェクトの変換時に、すべてのリンカオプションはデフォルト値に設定されます。XLINK オプションと ILINK オプションについて詳しくは、25 ページの *リンカオプションに関する違い*を参照してください。18 ページの *リンカとリンカの設定*を参照してください。

- **［出力コンバータ］**カテゴリ

  バージョン 4.x では、XLINK は数多くの出力フォーマットを生成でき、ユーザがリンカの**［出力］**ページで使用するものを指定します。バージョン 5.x の場合、ILINK は ELF/DWARF を生成します。ELF 出力を Intel-hex または Motorola S-records に変換する必要がある場合、**［出力コンバータ］**を使用してください。

- **［ライブラリビルダ］**カテゴリ

  バージョン 5.x には、新しいライブラリビルダがあります。つまり、自動的に変換されるオプションはありません。プロジェクトの変換時に、すべてのライブラリビルダオプションはデフォルト値に設定されます。

すべての新しいオプションを必ず設定してください。

IAR Embedded Workbench IDE で同等のオプションを設定する箇所については、『*IAR Embedded Workbench® IDE User Guide for ARM®*』を参照してください。

ランタイム環境とオブジェクトファイル

相互運用性

コンパイラのバージョン 5.x で生成されたコードをビルドするには、付属のランタイム環境コンポーネントを使用する必要があります。バージョン 5.x を使用して生成されたオブジェクトコードを、バージョン 4.x 付属のコンポーネントとリンクすることはできません。つまり、バージョン 4.x のオブジェクトコードをリビルドし、場合によってはソースコードを修正しなければならないこともあります。

アプリケーションを AEABI 準拠（ARM アーキテクチャの Embedded Application Binary Interface）にして、サードパーティ製リンカを用いて異なるベンダのオブジェクトファイルを含むアプリケーションをビルドできるようにするには、ツールで AEABI への準拠を有効にする必要があります。この方法について詳しくは、『*ARM 用 IAR C/C++ 開発ガイド*』を参照してください。

ランタイムライブラリファイルの選択

バージョン 4.x では、C/C++ の標準ライブラリが追加のサポートルーチンを含むビルド済ランタイムライブラリファイルとして提供されます。それぞれのファイルは特定のコンパイラオプションのセットを使用してビルドされています。

プロジェクトに使用するコンパイラオプションに応じて、プロジェクトオブジェクトに合ったランタイムライブラリファイルをコマンドラインで指定する必要があります。IAR Embedded Workbench IDE バージョン 4.x では、プロジェクト設定に基づいて正しいライブラリファイルが自動的に使用されます。

バージョン 5.x では、異なるグループのランタイムライブラリは提供されず、それぞれのグループは特定のコンパイラオプションのセットを使用してビルドされます。

プロジェクトに使用するコンパイラオプションに応じて、ILINK は正しい標準ライブラリファイルを自動的に使用します。必要があれば、コマンドラインで直接、あるいは IAR Embedded Workbench IDE バージョン 5.x で、ライブラリファイルを手動で指定することも可能です。

バージョン 5.x で使用できるライブラリファイルについて詳しくは、『*ARM 用 IAR C/C++ 開発ガイド*』を参照してください。

デバッガ

デバイス記述ファイル（ddf ファイル）のディレクトリ構造が変わりました。バージョン 5.x では、これらのファイルはデバイス固有のサブフォルダに配置されます。デバイス記述ファイルを明示的に選択する場合、新しいディレクトリ構造に注意する必要があります。

フラッシュローダ

アプリケーションのダウンロードにフラッシュローダを使用するには、simple-code フォーマットの追加の出力ファイルが 1 つ必要です。バージョンでは、手動で XLINK を設定して、この追加の sim ファイルを生成する必要があります。バージョン 5.x では、C-SPY がダウンロード用の情報を生成するため、この追加ファイルは必要ありません。

ツールオプション

このセクションでは、コンパイラやアセンブラ、リンカについて、バージョン 5.x とのコマンドラインオプションの違いを説明します。

コンパイラオプションに関する違い

次の表は、バージョン 4.x で変更されたコンパイラのコマンドラインオプションを示します。

古いコンパイラオプション	説明	バージョン 5.x
--library_module	ライブラリモジュールを作成	削除済み
--module_name	オブジェクトモジュール名を設定	削除済み
--omit_types	型情報を除外	削除済み
--segment	セグメント / セクション名を変更	--section
--separate_cluster_for_initialized_variables	初期化変数と非初期化変数を分離	削除済み
-s と -z	最適化レベルを設定	-O

表 4: コンパイラオプションの違い

-s と -z および -O の比較

バージョン 5.x では、最適化レベルを設定するコンパイラオプションの動作が、バージョン 4.x とは異なります。バージョン 4.x では、速度の最適化レベルを設定するオプションと、サイズの最適化レベルを設定するオプションが 1 つずつあり、どちらか一方を選択します。バージョン 5.x では、最適化レベ

ルを設定するオプションが 1 つあり、各レベルについてコンパイラがサイズと速度のバランスを調整します。最高レベルについては、サイズまたは速度に対して明示的に最適化を微調整することも可能です。

-O を -s および -z と比較した場合の対照表を以下に示します。

バージョン 4.x	バージョン 5.x
-s2、-z2	-On（なし）
-s3、-z3	-Ol（低）
-s6、-z6	-Om（中）
-z9	-Ohz（高、サイズ優先）
-s9	-Oh（高、速度とサイズのバランス）
--	-Ohs（高、速度優先）

表 5: コンパイラオプションの違い

注 : バージョン 5.x では、オプション -s と -z を使用すると、診断メッセージが出力されます。

アセンブラオプションに関する違い

次の表は、バージョン 4.x で変更されたアセンブラのコマンドラインオプションを示します。

古いアセンブラオプション	説明	バージョン 5.x
-b	ライブラリモジュールを作成	削除済み
-X	オブジェクトファイルで参照されていない外部のもの	削除済み

表 6: アセンブラオプションの違い

リンカオプションに関する違い

次の表は XLINK コマンドラインオプションと、対応する ILINK の機能の一覧です。

XLINK オプション	説明	ILINK
-!	コメント区切り文字	icf ファイルでは /*...*/ または //
-A	プログラムとしてロード	削除済み。16 ページの *アセンブラおよびアセンブラソースコード*を参照
-a	静的オーバレイを無効にする	削除済み
-B	出力ファイルを常に生成する	--force_output

表 7: XLINK オブジェクトに対応する ILINK の機能

XLINK オプション	説明	ILINK
-b	バンクセグメントを定義	icf ファイル*
-C	ライブラリとしてロード	削除済み。16 ページの *アセンブラおよびアセンブラソースコード*を参照
-c	プロセッサタイプを指定	削除済み
-D	シンボルを定義	--define_symbol
-d	コード生成を無効にする	削除済み
-E	オブジェクトコードを生成しない	削除済み
-e	外部シンボルの名前を変更する	--redirect
-F	出力フォーマットを指定	削除済み
-f	XCL ファイル名を指定	-f: ILINK では、設定ファイルはオプション --config を使用して指定
-G	グローバル型のチェックを無効にする	削除済み
-g	グローバルエントリを要求する	--keep
-H	未使用コードメモリをフィルする	--fill
-h	範囲をフィル	--fill
-I	インクルードパスを指定	削除済み
-J	チェックサムを生成する	--checksum
-K	コードを複製する	icf ファイル*
-L	リストファイルのディレクトリを指定する	--log_file
-l	指定ファイルにリスト作成	--log_file
-M	論理アドレスを物理アドレスにマッピングする	icf ファイル*
-n	ローカルのシンボルを無視	--no_locals
-O	複数の出力ファイル	削除済み
-o	出力ファイル	変更なし。ただし --output をエイリアスとして使用可

表 7: XLINK オブジェクトに対応する ILINK の機能（続き）

XLINK オプション	説明	ILINK
-P	パックされたセグメントを定義する	icf ファイル *
-p	1 ページあたりの行数を指定する	削除済み
-Q	分散ローディング	icf ファイル *
-q	リレー関数の最適化を無効化	削除済み
-R	アドレス範囲のチェックを無効にする	--diag_suppress
-r	デバッグ情報	削除済み。ILINK では、デバッグ情報はデフォルトでインクルードされ、--no_debug を使用して削除されます。
-rt	デバッグ情報（ターミナル I/O サポート）	--semihosting
-S	出力抑止操作	--silent
-s	新しいアプリケーションのエントリポイントを指定	--entry
-U	アドレス空間を共有する	icf ファイル *
-V	コードとデータ用に再配置エリアを宣言する	icf ファイル *
-w	診断制御を設定する	--diag_error、--diag_remark、--diag_suppress、--diag_warning、--diagnostics_tables、--error_list、--no_warnings、--remarks、--warnings_are_errors、--warnings_affect_exit_code
-x	クロスリファレンスリストを指定する	--map
-Y	Format variant	削除済み
-y	Format variant	削除済み
-Z	セグメントを定義する	icf ファイル *
-z	Segment overlap warnings	削除済み

表 7: XLINK オブジェクトに対応する ILINK の機能（続き）

* ILINK では、この機能はコマンドラインまたは IAR Embedded Workbench IDE で指定するリンカオプションとしては使用できません。代わりに、リンカ設定ファイルで指定する設定の一部となっています。

以下のオプションには大きな変更はありません。

`--image_input`、`--misrac`、`--misrac_verbose`

セグメントとセクションの比較

ここでは、バージョン 4.x のセグメントと、バージョン 5.x のセクションの違いについて説明します。

新しいセクション、セクション名、使用方法について詳しくは、『*ARM 用 IAR C/C++ 開発ガイド*』を参照してください。

命名規約

セグメントの命名規約は、セクションの場合とは少し異なります。

4.x では、すべてのセグメントを大文字で記述します。コードとデータセグメントについては、セグメントベース名がメモリ型を示します。また、静的データセグメントには内容の型を示すサフィックスも付いています。

5.x では、大文字で記述されるセクションと、先頭にピリオドの付いた小文字で表されるセクションがあります。これらのセクションの場合、セクション名は内容の型を表し、サフィックスはメモリ型を示します。

初期化用のセグメント

4.x では、コンパイラはイニシャライザ用と初期化済変数用に 1 つずつセグメントを作成します。5.x では、コンパイラはイニシャライザを含むデータセクションを 1 つ作成します。続いて、ILINK がこのセクションを初期化の正しい処理に変換します。初期化について詳しくは、『*ARM 用 IAR C/C++ 開発ガイド*』を参照してください。

旧式のセグメントから新しいセクションへのマッピング

旧式のセグメントの一部は完全に使用されなくなりました。新しいセクションの一部は、古いセグメントに対応していません。

次の表は旧式のセグメント名と、それに対応するバージョン 5.x のセクション名、および追加されたセクションを示します。

古いセグメント	新しいセクション	コメント
CODE	.text	
CODE_I	.textrw	
CODE_ID	.textrw	イニシャライザは独自のセクションを持たなくなりました。
CSTACK	CSTACK	
DATA_AC	--	定数の絶対配置はサポートされなくなりました。つまり、専用のセグメント / セクションはもう必要ありません。
DATA_AN	--	__no_init で宣言された絶対変数には空間を予約しなくなりました。すなわち、専用のセグメント / セクションはもう必要ありません。
DATA_C	.rodata	
DATA_I	.data	
DATA_ID	.data	イニシャライザは独自のセクションを持たなくなりました。
DATA_N	.noinit	
DATA_Z	.bss	
DIFUNCT	.difunct、PREDIFUNCT	
HEAP	HEAP	
ICODE	.text	
INITTAB	--	ILINK は異なる方法でこれを解決します。つまり、専用のセグメント / セクションは不要になりました。
INTVEC	.intvec	
IRQ_STACK	IRQ_STACK	
SWITAB	--	この機能は削除されました。専用のセグメント / セクションは不要になりました。

表 8: セグメントとセクションの対照表

アセンブラディレクティブ

一部のアセンブラディレクティブは削除されたか、動作が変わりました。次の表は、バージョン 5.x と 4.x で動作が異なるアセンブラディレクティブの一覧です。

バージョン 4.x のアセンブラディレクティブ	バージョン 5.x
ARGFRAME	削除済み
ASEG	削除済み
ASEGN	削除済み
BLOCK	削除済み
CFI	リソース名が標準化されました。CFI 名のブロックには、これらのリソース名のサブセットを含める必要があります。
COMMON	削除済み
DEFFN	削除済み
END	プログラム開始アドレスを引数として受け入れなくなりました。
ENDMOD	認識されますが効果はありません。ワーニングが出力されます。
FUNCALL	削除済み
FUNCTION	削除済み
LIBRARY	ライブラリモジュールを起動する代わりに、現在は ELF モジュールが起動します。構文が新しくなりました。
LIMIT	削除済み
LOCFRAME	削除済み
MODULE	ライブラリモジュールを起動する代わりに、現在は ELF モジュールが起動します。構文が新しくなりました。
MULTWEAK	削除済み
NAME	ELF プログラムモジュールを起動します。構文が新しくなりました。
ORG	削除済み
OVERLOAD	削除済み
PROGRAM	ELF プログラムモジュールを起動します。構文が新しくなりました。

表 9: アセンブラディレクティブの違い

バージョン 4.x のアセンブラディレクティブ	バージョン 5.x
RSEG	最初に使用される RSEG ディレクティブの先頭には、DC や DS、アセンブラ命令などなどのディレクティブを生成するコードを付けないでください。このディレクティブは、新しいディレクティブ SECTION のエイリアスになりました。構文が新しくなりました。
STACK	削除済み
SYMBOL	削除済み

表 9: アセンブラディレクティブの違い（続き）

バージョン 5.x のアセンブラディレクティブについては、『*ARM® IAR Assembler Reference Guide*』を参照してください。

ファイル名拡張子

次の表は、デフォルトのファイル名拡張子に関する違いの一覧です。

旧式のファイル名拡張子	新しいファイル名拡張子	説明 / コメント
s79	s	アセンブラソースファイル
r79	o	オブジェクトモジュール
r79	a	ライブラリオブジェクトモジュール
a79	out	ターゲットプログラム
d79	out	デバッグ用のターゲットプログラム

表 10: ファイル名拡張子の違い

ARM® IAR Embedded Workbench バージョン 4.x への移行

本ガイドでは、アプリケーションコードとプログラムのバージョン 4.x への移植のヒントを提供します。

ARM IAR C/C++ コンパイラのバージョン 3.x で最初に記述されたコードは、ARM IAR C/C++ コンパイラのバージョン 4.x でも使用できます。ただし、修正が必要な場合もあります。

このガイドでは、まず 2 つの製品バージョンの主な違いと、移行プロセスの概要について説明します。最後に、ARM IAR Embedded Workbench バージョン 3.x と 4.x の違いを解説します。製品間の違いと共通点について、簡単に説明します。

主な利点

このセクションでは、ARM IAR Embedded Workbench バージョン 3.x と比較した場合の、ARM IAR Embedded Workbench バージョン 4.x の主な利点について説明します。以降は新旧のバージョンを、*それぞれバージョン 3.x*、*バージョン 4.x* と表記します。

- オプションでタブグループとして整理できるドッキング可能なウィンドウにより効率的なウィンドウ管理
- シンボル定義を迅速に探しやすい関数、クラス、変数のカタログを持ったソースブラウザ
- スムーズな開発のスタートに役立つ、*設定不要*でプロジェクトをリンクおよび実行するためのテンプレートプロジェクト
- ビルドする構成が順番にリストされたバッチビルド
- 改善されてコンテキストに合った C/C++ ライブラリ関数のヘルプ
- 一般的なフラッシュダウンローダのフレームワーク
- コンパイラの最適化の改善により最大で 10% 小さいコードを実現
- EPI Majic JTAGインタフェースを使用したARM Embedded Trace Macrocellのサポート

- 新しいコプロセッサの組込み
- 構成が簡単な C/C++ ライブラリ
- ネスト割込みをサポートする新しいキーワード
- デバッグ時の STL コンテナのスマート表示
- デバッガウォッチウィンドウの自動表示
- 広範な機能拡張

移行についての考慮事項

古いプロジェクトを移行するために、以下の点を考慮してください。

- IAR Embedded Workbench IDE
- コードモデルおよびコードの生成
- プロジェクトオプション
- ランタイムライブラリとオブジェクトファイルの考慮事項

このリストにあるすべての項目がユーザのプロジェクトに適切なわけではありません。それぞれのケースにどのアクションが必要か、慎重に考えてください。

注 : バージョン 3.x 用に記述されたコードにより、バージョン 4.x でワーニングやエラーが出力される可能性がある点を認識しておくことが重要です。

IAR Embedded Workbench IDE

新しいバージョンの IAR Embedded Workbench IDE にアップグレードする際は、改良点が両バージョン間の互換性に大きく影響することはないため、スムーズに移行できるはずです。

ワークスペースとプロジェクト

3.x で作成したワークスペースおよびプロジェクトは、バージョン 4.x と互換性があります。プロジェクト設定に多少の違いがあります。このため、オプションを慎重に確認してください。詳細については、37 ページの*プロジェクトオプション*を参照してください。

C-SPYの レイアウトファイル

改善された新しいウィンドウ管理システムによって、3.x での C-SPY レイアウトファイルのサポートはなくなりました。カスタムの `lew` ファイルは、プロジェクトから問題なく削除できます。

コードモデルおよびコードの生成

バージョン 3.x では、ARM 用 IAR Embedded Workbench で 2 つの異なるコードモデルのどちらかを選択できました。小さいコードモデルは、Thumb モードで 4 MB、ARM モードで 32 MB にそれぞれ制限されていました。一方、大きいコードモデルでは、ARM コアでアドレス可能なメモリ空間を無制限に使用できました。ただし、大きいコードモデルでも、個々のリンカセグメントは 4 MB（Thumb モード）と 32 MB（ARM モード）に制限されていました。

具体的に指定されたセグメントへのコードの配置は、使用されるコードモデル、関数メモリの属性、プラグマセグメントディレクティブ、`--segment` コマンドラインオプションなど、多くの要因に左右されていました。

コンパイラ、特にリンカへの改善によって、2 つのコードモデルへの分割は廃止されました。この変更によって以下も影響を受けます。

- セグメント
- 拡張キーワード
- プラグマディレクティブ

セグメント

Thumb および ARM コードを含むセグメントパートは、同じセグメントに並んで配置されるようになりました。利点としては、よりコンパクトなコードの自動生成、無制限のセグメントサイズ、簡略化された設定プロセスなどが挙げられます。

直接の結果として、セグメントの命名が変わりました。これに伴ってバージョン 4.x に付属のリンカコマンドファイルも変更されましたが、カスタマイズされたリンカコマンドファイルを使用する場合、リンカコマンドファイルも同じように変更する必要があります。名前の変更は、データセグメントとコードセグメントにも影響する点に注意してください。

新旧のセグメント

次の表は、4.x より古いバージョンの IAR Embedded Workbench から移行する際に必要な、セグメントの移行をまとめたものです。

古いセグメント	新しいセグメント
FARFUNC_A	CODE
FARFUNC_T	CODE
NEARFUNC_A	CODE
NEARFUNC_T	CODE

表 11: 新旧のセグメント

古いセグメント	新しいセグメント
FARFUNC_A_I	CODE_I
FARFUNC_T_I	CODE_I
NEARFUNC_A_I	CODE_I
NEARFUNC_T_I	CODE_I
FARFUNC_A_ID	CODE_ID
FARFUNC_T_ID	CODE_ID
NEARFUNC_A_ID	CODE_ID
NEARFUNC_T_ID	CODE_ID
HUGE_AC	DATA_AC
HUGE_AN	DATA_AN
HUGE_C	DATA_C
HUGE_I	DATA_I
HUGE_ID	DATA_ID
HUGE_N	DATA_N
HUGE_Z	DATA_Z

表 11: 新旧のセグメント（続き）

注： _AN および _AC で終わるセグメントには、絶対アドレスに配置されたデータが含まれているため、リンカコマンドファイルには含めないでください。

拡張キーワード

コードモデルかデータモデルを選択することはなくなったため、すべての関数とデータメモリ属性が廃止されました。以下の廃止されたキーワードについては、コンパイラでワーニングメッセージが出力されます。

```
__farfunc、__nearfunc、__huge
```

プラグマディレクティブ

すべての関数およびデータメモリ属性が廃止されたことで、プラグマディレクティブに変更が発生しました。

```
#pragma segment="segment"
```

新しい構文については、『*ARM® IAR C/C++ Compiler Reference Guide*』を参照してください。

プロジェクトオプション

バージョン 4.x には、新しいプロジェクトオプションがいくつかあります。コマンドラインの派生形については、を参照してください。『*ARM® IAR C/C++ Compiler Reference Guide*』IAR Embedded Workbench の派生形については、『*IAR Embedded Workbench® IDE User Guide for ARM®*』を参照してください。

旧バージョンの ARM IAR Embedded Workbench から移行する際は、プロジェクトオプションへの以下の変更が特に重要です。

コマンドライン	IAR Embedded Workbench	説明
`--code_model`	コードモデル	廃止
`--segment`	セグメント名	構文の変更
`--no_tbaa`	型ベースエイリアス解析	新しい最適化。デフォルトで有効化

表 12: プロジェクトオプション

コードモデルかデータモデルを選択することはなくなったため、`--code_model` オプションが廃止されました。このため、このオプションが使用されると、コンパイラはワーニングメッセージを出力します。詳細については、35 ページの *コードモデルおよびコードの生成*を参照してください。

オプション `--segment` オプションの構文が変わりました。古い構文が使用されると、コンパイラはワーニングメッセージを出力します。

新しいコンパイラの最適化、［**型ベースエイリアス解析**］はデフォルトで有効になっています。この最適化は、オプション `--no_tbaa` か、IAR Embedded Workbench の同じ機能を使用して無効にすることができます。

ランタイムライブラリとオブジェクトファイルの考慮事項

コンパイラのバージョン 4.x で生成されたコードをビルドするには、付属のランタイム環境コンポーネントを使用する必要があります。バージョン 4.x を使用して生成されたオブジェクトコードを、バージョン 3.x 付属のコンポーネントとリンクすることはできません。

DLIBラ ンタイムライブラリによるコンパイルとリンク

旧バージョンでは、ランタイムライブラリを選択してもコンパイルに一切影響はありませんでした。ARM IAR Embedded Workbench バージョン 4.x では、ここが変わりました。ランタイムライブラリを構成して、アプリケーションで必要な機能を含めることが可能になりました。

入力と出力がその一例です。あるアプリケーションは fprintf 関数をターミナル I/O (stdout) に使用できますが、このアプリケーションはこのファイルに関連するファイル記述子にファイル I/O 機能を使用しません。この場合、ファイル I/O に関連するコードが削除されてもターミナル I/O 機能が得られるようにライブラリを設定できます。

この設定には、たとえば stdio.h のようなライブラリヘッダファイルが関係してきます。つまり、アプリケーションのビルド時に、ライブラリをビルドしたときと同じヘッダファイルの設定を使用しなければなりません。ライブラリ設定は *ライブラリ設定ファイル*で指定します。これはライブラリの機能を定義するヘッダファイルです。

IAR Embedded Workbench を使用してアプリケーションをビルドするときは、**[Normal]（通常）**、**Full（完全）**、**Custom（カスタム）**の 3 つのライブラリ設定から選択できます。**[Normal]** と **[Full]** は、製品に付属するビルド済のライブラリ設定です。上のファイル I/O の例では **[Normal]** を使用します。**[Custom]** は、カスタムでビルドされたライブラリで使用します。ライブラリ設定ファイルの選択は、自動的に処理される点に注意してください。

コマンドラインからアプリケーションをビルドする場合、ライブラリをビルドしたときと同じライブラリ設定ファイルを使用する必要があります。ビルド済のライブラリ (r79) については、対応するライブラリ設定ファイル (h) があり、これはライブラリと同じ名前です。これらのファイルは arm¥lib にあります。ライブラリ設定ファイルおよびライブラリオブジェクトファイルを指定するコマンドラインは、以下のようになります。

```
iccarm -D_DLIB_CONFIG_FILE=...¥arm¥lib dl4tpainl8n.h
xlink dl4tpainl8n.r79
```

独自のライブラリバージョンをビルドする場合は、デフォルトのライブラリ設定ファイル dlArmCustom.h を使用してください。

この機能を活用するために、『*ARM® IAR C/C++ Compiler Reference Guide*』のランタイム環境の箇所に目を通すことをお勧めします。

PROGRAM ENTRY

デフォルトでは、リンカにはアプリケーションのビルド時にプログラムモジュール内で root により宣言されたセグメントパートがすべて含まれています。ただし、ロードの手順に影響する新しいメカニズムがあります。

*開始ラベル*を指定するための、新しいリンカオプション［**エントリラベル**］(-s) があります。開始ラベルを指定することで、リンカは一致する開始ラベルがあるかどうかすべてのモジュールを検索し、そのポイントからロードを開始します。以前と同じように、ルートセグメントパートを含むすべてのプログラムモジュールもロードされます。

バージョン 4.x では、cstartup.s79 のデフォルトのプログラムエントリラベルは __program_start です。つまり、リンカがそこからロードを開始することになります。この新しい動作の利点は、cstartup.s79 の一時変更がずっと簡単になることです。

IAR Embedded Workbench でアプリケーションをビルドする場合、カスタマイズした cstartup ファイルを自分のプログラムに追加します。そのファイルがライブラリで cstartup モジュールの代わりに使用されます。プログラムエントリポイントの名前を一時変更することで、起動ファイルを切り替えることも可能です。

コマンドラインからアプリケーションをビルドする場合、C/C++ アプリケーションをリンクする際に -s オプションを明示的に指定する必要があります。このオプションなしにリンクする場合、結果として出力される実行可能ファイルは参照先のモジュールがないために空になります。

システム初期化 — CSTARTUP

cstartup.s79 ファイルの内容が、次の 3 つのファイルに分割されました。

```
cstartup.s79、cmain.s79、cexit.s79
```

現在は cstartup.s79 には、設定スタックとプロセッサモードに対する例外ベクタと初期起動コードのみが含まれます。cstartup.s79 ファイルは、3 つの中で修正を必要とする可能性がある唯一のファイルです。

cmain.s79 ファイルは、データセグメントを初期化して C++ コンストラクタを実行します。cexit.s79 ファイルには、C++ デスクトラクタの実行などの終了コードが含まれます。

cstartup.s79 の修正したコピーを使用するアプリケーションは、新しいファイル構造に適合させる必要があります。

CSTARTUPリ セットベクタ

バージョン 3.x より前は、cstartup.s79 のリセットベクタは ?cstartup の最初の命令に対する相対的な分岐として実装されていました。

```
B ?cstartup
```

この方法の欠点は、B 命令の分岐範囲が 32 MB に限定されていることでした。特にデバッグ時には、アドレスマップの上部にあるメモリにコードを配置することがかなり一般的です。すなわち、コードは B 分岐命令の及ばないところに配置されることになります。

この制限を取り除くために、分岐命令ではなく絶対ジャンプを行えるように 3.x が変更されました。

```
LDR PC,=?cstartup
```

追加の定数値を受け入れるために、予約済のベクタエリアが 32 バイトから 64 バイトに増加されました。この解決方法によって、リセットベクタは 4 GB のアドレス空間内のどこにでも届くようになりました。この方法は通常、アプリケーションのデバッグ時に問題なく機能します。

ただ、この方法で問題が発生する場合が 2 つあります。

- ゼロ位置のフラッシュメモリにプログラムをダウンロードし、それが別の位置に後で再配置される
- コードがコピーされて RAM 内で実行される

ゼロ位置にリセットベクタがあって、コードがアドレス `0x100000` に再配置されるアプリケーションの場合、アプリケーションはそのアドレスに配置されているという前提のもとにリンクされます。リセット時に、アドレス `0x100000` の内容はゼロ位置のアドレスにあります。リセットベクタに `0x100100` といった絶対ジャンプが含まれるため、このジャンプは失敗します。これは、コードがまだそのアドレスにマッピングまたはコピーされていないことが理由です。この場合は、相対分岐命令であれば機能します。

アプリケーションで再配置またはコピーのメカニズムを使用する場合、`cstartup.s79` ファイルを修正して、デフォルトの絶対ジャンプではなく相対ジャンプを実行するようにリセットベクタを変更する必要があります。

デバイス固有のヘッダファイル

周辺レジスタを定義するヘッダファイルの一部で、レジスタの命名規約が変わりました。レジスタ名の先頭に付く下線 2 つ (__) が廃止されました。下位互換性のために、旧式のファイルのコピーが製品に同梱されています。たとえば、旧バージョンの `iolpc210x.h` は現在、`iolpc210x_old.h` として利用できます。