从 XLINK 迁移到 ILINK 时的注意要点

背景介绍

有些用户由于各种原因需要将其应用从 IAR EWARM 4.x 版迁移到 IAR EWARM 5.x 及以后的版本，因为 IAR 这两个版本之间的变化非常大，其链接器从 XLINK 变成了 ILINK，文件格式已经完全不一样了，因此需要了解其差异，以便顺利完成迁移。本文这种介绍了迁移过程要注意的一些问题。

1. 迁移过程要考虑的问题

- 使现有的应用程序源代码编译和链接成功
- 确定运行时行为的潜在变化

2. 迁移过程需要注意如下变化

- 编译器和 C 源代码
- 汇编程序和汇编程序源代码
- 链接器和链接器配置
- 运行时环境和目标文件
- 工程项目配置文件

本文我们主要介绍**链接器和链接器配置，**迁移过程中，除了需要更改链接器配置文件之外，可能还需要根据应用程序的具体情况，对某些其它方面做小的修改，如 C/C++源代码，汇编语言源代码，运行时环境和目标文件和工程项目配置文件，必要时可以参考 IAR Embedded Workbench for ARM 5.xx 以后的所带的 EWARM_MigrationGuide.pdf 手册。。

3. 链接器和链接器的配置

IAR XLINK 链接器已经被 IAR ILINK 链接器代替了。

3.1 XLINK 与 ILINK 对比

IAR EWARM 4.xx使用的是IAR的XLINK链接器，而IAR EWARM 5.xx以后的版本使用的是ILINK链接器，它们的主要区别是建立目标代码所用的文件格式不同。在IAR EWARM 4.xx版本下，使用的是IAR私有的UBROF格式，而IAR EWARM 5.xx版本下使用的是业界标准格式ELF/DWARF。遵循ARM公司提出的ABI（Application Binary Interface）标准，IAR EWARM 5.xx提供了目标文件级别的兼容性，即其它ABI兼容工具生成的目标库可以与EWARM生成的目标文件一起链接并调试；同时EWARM生成的目标库也能在其它ABI兼容工具里参与链接和调试，使得应用程序的开发更具灵活性。当然，这也意味着EWARM 5.xx里使用了全新版本的链接器ILINK来取代原先所用的XLINK，从而导致链接器配置文件也使用了新的格式: ICF，而不再是原先的XCL。

3.2 从 XLINK 到 ILINK 的迁移

1. 新的 IAR ILINK 链接器是目标相关的，它取代了 IAR XLINK 链接器； 可执行文件的名称已从 xlink 重命名为 ilinkarm。
2. 要将链接器命令文件 xcl 迁移到新的 ILINK 配置文件 icf
3. 将 XLINK 的段映射到新的 ILINK 的段
4. XLINK 的 xcl 到 ILINK 的 icf 迁移文件的一个示例

3.3 将 XLINK.XCL 转换为 ILINK.ICF

由于链接器命令文件（XLINK）和链接器配置文件（ILINK）基于两个不同的范例，因此链接器命令文件中的任何内容均不会自动转换。相反，您必须手动转换链接器设置。

如果使用的是 IAR Embedded Workbench IDE，则可以使用链接器配置文件编辑器来设置链接器配置。

1. 选择 **Project**-> **Options**，选择 **Linker** 类别，然后单击 **Config** 选项卡。
2. 要打开链接器配置文件编辑器，请选择 **Override default** 选项，然后单击 **Edit** 按钮。
3. 在出现的对话框中，您可以定义：
   - 中断向量表的起始地址
   - RAM 和 ROM 存储器区域的开始和结束地址
   - 堆栈和堆的大小
4. 完成后，单击 **Save** 按钮。首次执行此操作时，将出现 **Save As** 对话框。

注意：您必须为所有构建配置明确选择专用的链接器配置文件。

如果从命令行构建项目，则可以使用位于 arm\config 目录中的链接器配置文件 generic.icf 或位于 examples 目录中的示例项目中可用的任何配置文件。 您可以将这些配置文件中的任何一个用作模板，以创建适合您的目标硬件和应用程序需求的配置文件。

要想完成迁移我们先要了解 XLINK 和 ILINK 的一些知识。

3.4 XLINK 的相关知识

XLINK 的段类型

1. CODE 可执行代码
2. CONST 存储在 ROM 中的数据
3. DATA 存储在 RAM 中的数据

XLINK 的段和其功能说明

段	描 述	段类型	属性
CODE	保存将在 ROM 中执行的程序代码，系统初始化代码和__ramfunc 关键	CODE	只读

	字声明的代码除外		
CODE_I	保存声明为__ramfunc 的程序代码，在 RAM 中执行	DATA	读/写
CODE_ID	用于初始化 CODE_I 的代码	CONST	只读
DATA_C	保存常数数据，包括文字字符串	CONST	只读
DATA_I	保存用非 0 值初始化的静态和全局变量	DATA	读/写
DATA_ID	保存位于 DATA_I 段的静态和局部变量的初值	CONST	只读
DATA_N	保存位于非易失性存储器中用关键字__no_init 声明的静态和全局变量	DATA	读/写
DATA_Z	保存无初始值或用 0 初值声明的静态和全局变量。变量由启动代码在初始化期间清 0	DATA	读/写
DIFUNCT	C++所要求的动态初始化代码	CODE	只读
SWITAB	保存软件中断向量表	CODE	只读
INITAB	保存启动后初始化期间所需要的段地址和段长度表	CONST	只读
INTVEC	保存复位与异常向量	CODE	只读
ICODE	保存启动代码	CODE	只读
CSTACK	User 和 System 模式所用到的栈	DATA	读/写
IRQ_STACK	用于保存 IRQ 异常服务的堆栈。	DATA	读/写
HEAP	保存动态分配的数据	DATA	读/写

XLINK 链接器的一些命令的简单说明

最常用的命令选项有：CPU 命令选项-c、常数定义命令选项-D、段定位命令选项-Z 或-P。

“-c”命令选项用于规定用户系统所采用的 CPU，如：

-carm

“-D”命令选项用于规定存储器的起始和终止地址，如：

-DROMSTART=40000040

-DROMEND=40006FFF

“-D”命令选项也可用于定义堆栈度或其他常数，如：

-D_CSTACK——STZE=2048

-D_IRQ_STACK_SIZE=5I2

“-Z”命令选项按段出现的顺序进行定位，对每个存储器范围要指定其终点，如：

-Z(CONST)MYSEGMENTA，MYSEGMENTB=008000-OFFFFF

两个不同类型的段如果不指定第 2 个段的范围，则可以定位在同一个存储器区域之内，如：

-Z(CONST)MYSEGMENTA=008000-0FFFFF

-Z(CODE)MYCODE

两段存储器范围可以覆盖，从而允许具有不同定位要求的段共享部分存储器空间，如：

-Z(CONST)MYSMALLSEGMENT=008000-000FFF

-Z(CONST)MYLARGESEGMENT=008000-OFFFFF

“-P”命令选项以非连续方式进行段定位，可充分利用存储器空间，如：

-P(DATA)MYDATA=100000-101FFF，110000-11IFFF

如果用户应用系统还有一段 RAM 位于存储器 0x10F000 ~ OxI0F7FF，

只要将这段范围加到上述命令中即可：

-P(DATA)MYDATA=100000-10IFFF，10F00010F7FF，110000-11IFFF

3.5 ILINK 的相关知识

ILINK 的主要段和块及其功能的说明

Section 名称	描述	存储空间
CSTACK	User 和 System 模式所用到的栈	RAM
IRQ_STACK	IRQ 模式所用到的栈	RAM
HEAP	堆	RAM
.intvec	异常向量表	ROM
.cstart	初始化代码	ROM
.text	程序代码	ROM
.data	初始化的静态和全局变量	RAM
.bss	未初始化的静态和全局变量	RAM
.noinit	由__no_init 声明的静态和全局变量	RAM
.rodata	常量	ROM

ILINK 链接器的一些常用命令的简单说明

define [exported] symbol *name = expr;*
作用： 指定某个符号的值。
参数:
exported 导出该 symbol，使其对可执行镜像可用
name 符号名
expr 符号值
举例:

```
define symbol RAM_START_ADDRESS      = 0x40000000;
define symbol RAM_END_ADDRESS        = 0x4000FFFF;
```

define memory name with size = expr [*, unit-size*];
作用： 定义一个可编址的存储地址空间（memory）。
参数:
name memory 的名称
expr 地址空间的大小
unit-size expr 的单位，可以是位（unitbitsize），缺省是字节（unitbytesize）
举例:

```
define memory MEM with size = 4G;
```

define region *name = region-expr;*

作用： 定义一个存储地址区域（region） 。一个区域可由一个或多个范围组成，每个范围内地址必须连续，

但几个范围之间不必是连续的。

参数：

name　　region 的名称

region-expr　　*memory*:[from *expr* { *to expr* | size *expr*}]，可以定义起止范围，也可以定义起始地址和 region 的大小

举例：

define region ROM = MEM:[from 0x0 size 0x10000];

define region ROM = MEM:[from 0x0 to 0xFFFF];

define block *name*[with *param, param...*]

{

extended-selectors

};

作用：　定义一个地址块（block）；它可以是个空块，比如栈、堆；也可以包含一系列 sections。

参数：

name　　block 的名称

param　　可以是：　　size = expr　　（块的大小）

maximum size = expr　　（块大小的上限）

alignment = expr　　（最小对齐字节数）

fixed order　　（按照固定顺序放置 sections）

extended-selector　　[first | last] { *section-selector* | block *name* | overlay *name* }

first　　最先存放

last　　最后存放

section-selector　　[*section-attribute*][section *sectionname*][object *filename*]

section-attribute　　[readonly [code | data] | readwrite [code | data] | zeroinit]

sectionname　　section 的名称

filename　　目标文件的名称

即可以按照 section 的属性、名称及其所在目标文件的名称这三个过滤条件中，任意选取一个条件，或选取多个条件进行组合，来圈定所要求的 sections。

name　　block 或 overlay 的名称

举例：

define block HEAP with size = 0x1000, alignment = 4 { };

define block MYBLOCK1 = { section mysection1, section mysection2, readwrite };

define block MYBLOCK2 = { readonly object myfile2.o };

initialize { by copy | manually } [with *param, param...*]

{

section-selectors

};

作用：　初始化 sections。

参数:

by copy	在程序启动时自动执行初始化。
manually	在程序启动时不自动执行初始化。
param	可以是: packing = { none | compress1 | compress2 | auto } copy routine = *functionname* packing 表示是否压缩数据，缺省是 auto。 functionname 表示是否使用自己的拷贝函数来取代缺省函数。
section-selector	同前

举例:

initialize by copy { rw };

do not initialize
{
section-selectors
};

作用: 规定在程序启动时不需要初始化的 sections。一般用于__no_init 声明的变量段（.noinit）。

参数:

section-selector	同前

举例:

do not initialize { .noinit };

place at { address *memory*[: *expr*] | start of *region_expr* | end of *region_expr* }
{
extended-selectors
};

作用: 把一系列 sections 和 blocks 放置在某个具体的地址，或者一个 region 的开始或者结束处。

参数:

memory	memory 的名称
expr	地址值，该地址必须在 memory 所定义的范围内
region_expr	region 的名称
extended-selector	同前

举例:

place at start of ROM { section .cstart };
place at end of ROM { section .checksum };
place at address MEM:0x0 { section .intvec };

place in *region-expr*
{
extended-selectors
};

作用： 把一系列 sections 和 blocks 放置在某个 region 中。sections 和 blocks 将按任意顺序放置。

参数：

region-expr region 的名称

extended-selector 同前

举例：

```
place in ROM { readonly };                          /* all readonly sections */
place in RAM { readwrite };                         /* all readwrite sections */
place in RAM { block HEAP, block CSTACK, block IRQ_STACK };
place in ROM { section .text object myfile.o };     /* the .text section of myfile.o */
place in ROM { readonly object myfile.o };          /* all read-only sections of myfile.o */
place in ROM { readonly data object myfile.o };     /* all read-only data sections myfile.o */
```

3.6 XLINK 的段对应的 ILINK 的段

XLINK 段	ILINK 段	注释
CODE	.text	
CODE_I	.textrw	
CODE_ID	.textrw	初始化程序不再具有自己的段。
CSTACK	CSTACK	
DATA_AC	--	不再支持常量的绝对放置，不再需要专用的段。
DATA_AN	--	__no_init 声明的绝对变量不再保留空间，不再需要专用的段。
DATA_C	.rodata	
DATA_I	.data	
DATA_ID	.data	初始化程序不再具有自己的段。
DATA_N	.noinit	
DATA_Z	.bss	
DIFUNCT	.difunct, PREDIFUNCT	
HEAP	HEAP	
ICODE	.text	
INITTAB	--	ILINK 使用另一种方法来解决此问题，不再需要专用的段。

INTVEC	.intvec	
IRQ_STACK	IRQ_STACK	
SWITAB	--	此功能已被删除，不再需要专用的段。

3.7 XLINK 的 xcl 到 ILINK 的 icf 迁移文件示例

XLINK 的 xcl 文件例子

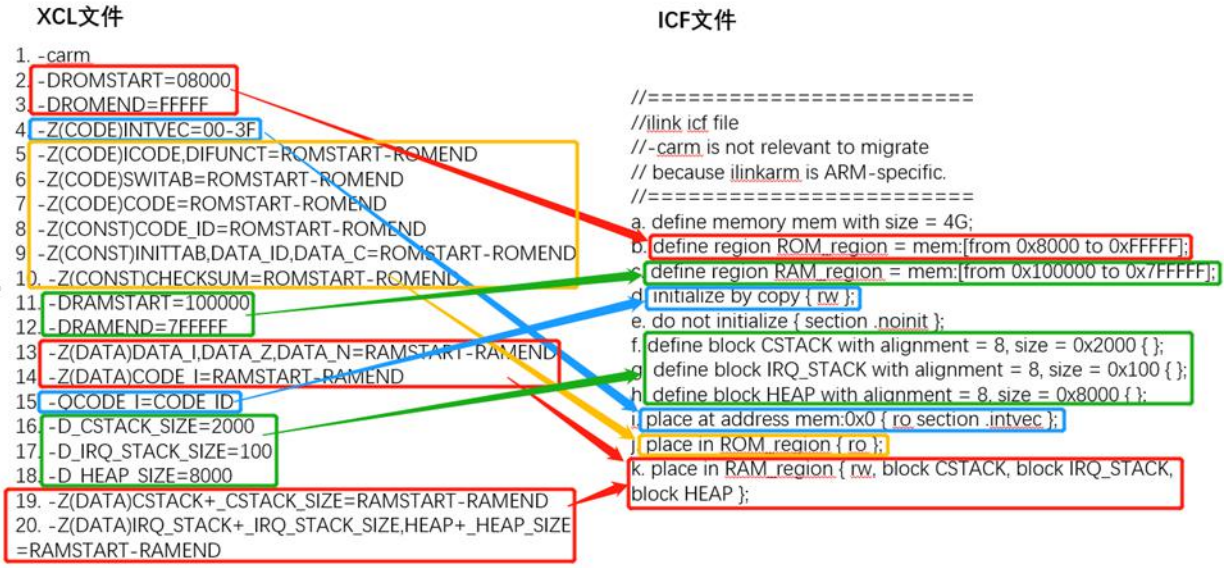
```
-!=========================
-!xlink xcl file
-!=========================
-carm
-DROMSTART=08000
-DROMEND=FFFFF
-Z(CODE)INTVEC=00-3F
-Z(CODE)ICODE,DIFUNCT=ROMSTART-ROMEND
-Z(CODE)SWITAB=ROMSTART-ROMEND
-Z(CODE)CODE=ROMSTART-ROMEND
-Z(CONST)CODE_ID=ROMSTART-ROMEND
-Z(CONST)INITTAB,DATA_ID,DATA_C=ROMSTART-ROMEND
-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
```

对应的 ILINK 的 icf 文件

```
//=========================
//ilink icf file
//-carm is not relevant to migrate because ilinkarm is ARM-specific.
//=========================
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 };
```

下面是一个 XLINK 链接器命令文件（xcl）转换为一个的 ILINK 链接器配置文件（icf）的对应关系图。



4. 其它

在 EWARM 5.xx 以后的版本中，默认的程序入口符号（Program Entry）由原先的__program_start 更改为__iar_program_start，因此对于旧的 4.xx 汇编代码而言，需要更改这个入口符号名；当然也可以在 EWARM 5.xx 以后的 Linker 配置选项中修改默认的 Program Entry。

如果在 EWARM 5.xx 版本中直接打开 4.xx 所创建的工程文件，会有对话框询问是否自动将其转换成 5.xx 的工程文件；若选择 OK，4.xx 的工程文件会被转换成 5.xx 的工程文件，当然原来的 4.xx 工程文件也会自动生成一个备份。某些配置信息无法被自动带入 5.xx 的工程，如链接器配置文件的路径等，因此请仔细检查相关的编译、汇编或链接选项，确保它们具有正确的设置。如在 EWARM 6.xx/7.xx/8.xx 中，由于差异较大，可能需要您重新创建工程项目，然后对照 EWARM 4.xx 的工程项目的配置设置其设置新的选项。

5. 附录

本文编译自 IAREWARM_MigrationGuide.ENU 手册。