78K IAR Embedded Workbench®

Migration Guide

for NEC Electronics'

78K0 and 78K0S Microcontroller Subfamilies

EDITION NOTICE

Second edition: May 2005

Part number: M78K-2

This guide applies to version 4.x of the 78K IAR Embedded Workbench®.

Contents

Migrating to the 78K IAR Embedded Workbench version 4.x

This guide gives hints for porting your application code and projects to the 78K IAR Embedded Workbench IDE version 4.x.

C source code that was originally written for the IAR 78000 C Compiler version 3.x can be used also with the new 78K IAR C/C++ Compiler version 4.x. However, some modifications may be required.

This guide presents the major differences between the 78K IAR Embedded Workbench version 3.x and the 78K IAR Embedded Workbench version 4.x, and describes the migration considerations.

Key advantages

This section lists the major differences between the IAR 78000 C Compiler version 3.x and the 78K IAR C/C++ Compiler version 4.x. Hereafter, the two compiler versions are referred to as *version 3.x* and *version* 4.x, respectively.

- The most obvious difference is that support for C++ has become available.
- Moreover, 4.x adheres more strictly to the ISO/ANSI C standard; for example, it is possible to use pragma directives instead of extended keywords for defining special function registers (SFRs).
- The checking of data types adheres more strictly to the ISO/ANSI C standard, compared to version 3.x. This helps you to identify and correct problems in the code, which improves the quality of the object code.
- Efficient window management through dockable windows optionally organized in tab groups.
- Source browser with a catalog of functions, classes, variables, et cetera, for a quick navigation to symbol definitions.
- Template projects to get a project that links and runs *out of the box* for a smooth development start up.
- Batch build with ordered lists of configurations to build.
- Improved context-sensitive help with reference information for windows, dialog boxes, and C/C++ DLIB library functions.

- Easy configuration of the C/C++ libraries.
- Smart display of STL containers at debugging
- Auto-display debugger watch window
- A broad range of small feature enhancements to improve the look and feel.

Note: It is important to be aware of the fact that code written for version 3.x may generate warnings or errors in version 4.x.

The migration process

In short, to migrate to version 4.x, consider the following:

- The project file and project setup
- C source code and compiler considerations
- Assembler considerations
- Runtime environment, runtime library, and object file considerations.

To migrate your old project, follow the described migration process. Note that not all items in the described migration process may be relevant for your project. Consider carefully what actions are needed in your case.

Project file and project setup

If you are using the IAR Embedded Workbench IDE, follow these steps to verify that your project file has been properly converted:

1. Start your new version of the 78K IAR Embedded Workbench and create a new workspace by choosing **File>New** and then **Workspace**.
2. Choose **Project>Add Existing Project** to insert your old project into the workspace. This step will create two new project files with the same name as the old file, but with the extensions `ewp` and `ewd`. The `ewp` file contains all settings required to build the application, while the `ewd` file contains all settings related to the debugger. The old project file will remain untouched.
3. It is strongly recommended that you verify that your options have been setup correctly.

   To generate a text file with the command line equivalents of the project options in your old project, see *Migrating project options*, page 8.

   Also, set any new options.
4. If you have your own linker command file, compare this file with the original file in the old installation and make the required changes in a copy of the corresponding file in the new installation. For information about changes related to segments, see *Segments*, page 19.

C source code and compiler considerations

In short, the process of migrating from version 3.x to version 4.x involves the following steps:

1. Replace or modify extended keywords according to the description in the section *Extended keywords*, page 12. The include file for the old intrinsic functions, `in78k.h`, defines some of the old extended keywords and can be used if you want to keep the old syntax. However, the syntax of the keywords for interrupts must be migrated.

2. Replace or modify pragma directives according to the section *Pragma directives*, page 16.

3. Make sure not to use nested comments in your source code. In version 4.x, nested comments are never allowed. For more information, see *Nested comments*, page 24.

4. The new compiler uses a different C parser and a large number of new optimizations have been added. Depending on your old source code, this might require you to modify your source code. One example of this is a simple delay loop, such as:

   ```
   i = 50000;
   do {i--;}
   while (i-- != 0);
   ```

   This code will be removed by the optimizer, unless you declare the variable `i` as `volatile`.

   In order to produce more efficient code, the compiler performs transformations like, for example, removing redundant calculations, replacing division by shift and removing useless calculations. Code that the compiler considers as *not useful* is removed and this may cause unexpected effects as in this example.

5. Replace or modify intrinsic functions according to the section *Intrinsic functions*, page 18.

6. Version 4.x will by default not accept preprocessor expressions containing any of the following:

   - Floating-point expressions
   - Basic type names and `sizeof`
   - All symbol names (including typedefs and variables).

   With the option `--migration_preprocessor_extensions`, version 4.x will accept such non-standard expressions. For details about this option, see the *78K IAR C/C++ Compiler Reference Guide*.

Assembler considerations

- If your application is written partly in assembler and partly in C, and if you have used any of the memory segments specific to version 3.x in assembler source code, you must replace all old segment names with new segment names. For further details, see the section *Segments*, page 19.
- The old assembler environment variables have changed:

Version 3.x assembler	Version 4.x assembler
ASM_78000	ASM_78K
A78000_INC	A78K_INC

- If your application is written entirely in assembler, you should not include a library in your application. To exclude the library from the build, choose **Project>Options**, select the **General Options** category and click the **Library Configuration** tab. Select **None** from the **Library** drop-down list.
- The RSEG directive is no longer allowed in MACRO definitions, because it causes calculations of relative jump distances to become incorrect.

Runtime library, runtime environment, and object file considerations

In version 4.x, two sets of runtime libraries are provided—the IAR DLIB Library and the IAR CLIB Library. CLIB corresponds to the runtime library provided with version 3.x, and it can be used in the same way as before.

For information about how to migrate from the CLIB library to the DLIB library, see *Migrating from CLIB to DLIB*, page 6. For more information about the two libraries, and the runtime environment they provide see the *78K IAR C/C++ Compiler Reference Guide*.

To build code produced by version 4.x of the compiler, you must use the runtime environment components it provides. For information about how to link object code produced using version 4.x with components provided with version 3.x, see *Object file considerations*, page 7.

COMPILING AND LINKING WITH THE DLIB RUNTIME LIBRARY

In earlier versions, the choice of runtime library did not have any impact on the compilation. In 78K IAR Embedded Workbench version 4.x, this has changed. Now you can configure the runtime library to contain the features that are needed by your application.

One example is input and output. An application may use the `fprintf` function for terminal I/O (`stdout`), but the application does not use file I/O functionality on file descriptors associated with the files. In this case the library can be configured so that code related to file I/O is removed but still provides terminal I/O functionality.

This configuration involves the library header files, for example `stdio.h`. This means that when you build your application, the same header file setup must be used as when the library was built. The library setup is specified in a *library configuration file*, which is a header file that defines the library functionality.

When building an application using the IAR Embedded Workbench, there are three library configuration alternatives to choose between: **Normal**, **Full**, and **Custom**. **Normal** and **Full** are prebuilt library configurations delivered with the product, where **Normal** should be used in the above example with file I/O. **Custom** is used for custom built libraries. Note that the choice of the library configuration file is handled automatically.

When building an application from the command line, you must use the same library configuration file as when the library was built. For the prebuilt libraries (`r26`) there is a corresponding library configuration file (`h`), which has the same name as the library. The files are located in the `78k\lib` directory. The command lines for specifying the library configuration file and library object file could look like this:

```
icc78k -D_DLIB_CONFIG_FILE=C:\...\78k\lib\dlib\dl78ksln.h
xlink dl78ksln.r26
```

In case you intend to build your own library version, use the default library configuration file `dl78kCustom.h`.

To take advantage of the features it is recommended that you read about the runtime environment in the *78K IAR C/C++ Compiler Reference Guide*.

PROGRAM ENTRY

By default, the linker includes all `root` declared segment parts in program modules when building an application. However, there is a new mechanism that affects the load procedure.

There is a new linker option **Entry label** (`-s`) to specify a *start label*. By specifying the start label, the linker will look in all modules for a matching start label, and start loading from that point. Like before, any program modules containing a root segment part will also be loaded.

In version 4.x, the default program entry label in `cstartup.s26` is `__program_start`, which means the linker will start loading from there. The advantage of this new behavior is that it is much easier to override `cstartup.s26`.

If you build your application in the IAR Embedded Workbench, just add your customized `cstartup` file to your project. It will then be used instead of the `cstartup` module in the library. It is also possible to switch startup files just by overriding the name of the program entry point.

If you build your application from the command line, the `-s` option must be explicitly specified when linking a C/C++ application. If you link without the option, the resulting output executable will be empty because no modules will be referred to.

SYSTEM INITIALIZATION—CSTARTUP

The content of the `cstartup.s26` file has been split up into two files:

`cstartup.s26`, `cexit_clib.s26` (for CLIB), `cexit_dlib.s26` (for DLIB)

Now, the `cstartup.s26` file contains exception vectors and initial startup code to setup stacks and it initializes data segments and executes C++ constructors. Note that only the `cstartup.s26` file might require any modifications.

The `cexit.s26` file contains termination code, for example, execution of C++ destructors.

For old applications that used a modified copy of `cstartup.s26`, you must make a copy of the supplied new `cstartup` file and adapt it to your needs.

MIGRATING FROM CLIB TO DLIB

There are some considerations to have in mind when if you want to migrate from the CLIB, the legacy C library, to the modern DLIB C/C++ library:

- The CLIB `exp10()` function defined in `iccext.h` is not available in DLIB.
- The DLIB library uses the low-level I/O routines `__write` and `__read` instead of `putchar` and `getchar`.
- If the heap size in your version 3.x project using CLIB was defined in a file named `heap.c`, you must now set the heap size either in the extended linker command file (`*.xcl`) or in the Embedded Workbench to use the DLIB library.

You should also see the chapter *The DLIB runtime environment* in the *78K IAR C/C++ Compiler Reference Guide*.

DEVICE-SPECIFIC HEADER FILES

The header files that defines peripheral registers delivered with version 2.x and 3.x cannot be used with version 4.x. For version 4.x applications, make sure to use the header files delivered with version 4.x.

CALLING CONVENTION

For backwards compatibility, the 78K IAR C/C++ Compiler version 4.x also supports the calling conventions used by the versions 2.x and 3.x of the compiler. For information about the old calling conventions, see the user documentation provided with those compiler versions. To use any of the old calling conventions in version 4.x, define and declare your functions with the appropriate keyword available for backward compatibility:

Keywords for backward compatibility	Description
`__V2_call`	Calling convention compatible with version 2.x of the compiler
`__V3_call`	Calling convention compatible with version 3.x of the compiler

Table 1: Keywords for old calling conventions

OBJECT FILE CONSIDERATIONS

Object files (`r26`) created with the compiler version 3.x or 2.x will be possible to link with object files created with version 4.x under the following conditions:

- The appropriate extended keyword `__V3_call` or `__V2_call` is used in function declarations
- The object file does not call any CLIB library functions. This is not allowed because the calling convention has changed
- Debug information cannot be available
- The version 4.x `cstartup.s26` file has been modified, see *System initialization—Cstartup*, page 6
- The linker command file has been updated with segment name information, see *Compiler options*, page 8.

Linker errors and warnings

When linking the application, the following linker error might be generated:

```
Error [e46]: Undefined external "?xxxxx_Lnn" referred in Example1
```

This error message is generated if the version 3.x or 2.x object module is referring a runtime library function which is not available in the version 4.x runtime library. In this case, you can extract the referred function from the version 3.x or 2.x runtime library module and include it in your version 4.x project by using the IAR XLIB Librarian.

The following linker warnings can be ignored:

```
Warning [w31]: Modules have been compiled with possibly
incompatible settings:
1 modules (including 'Example1' [78000 v1 m0 ...]) used one set.
```

```
1 modules (including 'Example2' [78000 v1 m0 ...]) used one set.

Warning [w6]: Type conflict for external/entry "Example3", in
module Example4 against external/entry in module Example5; types
have different memory attributes
```

Compiler options

The command line options in version 4.x follow two different syntax styles:

- Long option names containing one or more words prefixed with two dashes, and sometimes followed by an equal sign and a modifier, for example `--strict_ansi` and `--module_name=test`.
- Short option names consisting of a single letter prefixed with a single dash, and sometimes followed by a modifier, for example `-r`.

Some options appear in one style only, while other options appear as synonyms in both styles. A number of new command line options have been added. For a complete list of the available command line options, see the *78K IAR C/C++ Compiler Reference Guide*.

The old environment variable `QCC78000` has changed to `QCC78K`.

MIGRATING PROJECT OPTIONS

Since the available compiler options differ between version 3.x and version 4.x, you should verify your option settings after you have converted an old project.

If you are using the command line interface, you can simply compare your makefile with the option tables in this section, and modify the makefile accordingly.

If you are using the IAR Embedded Workbench IDE, all option settings are automatically converted during the project conversion.

However, it is still recommended to verify the options manually. Follow these steps:

1. Open the old project in the old IAR Embedded Workbench version.
2. In the project window, select the project level to get information about options on all levels in your project.
3. To save the project settings to a file, right-click in the project window. On the context menu that appears, choose **Save As Text**, and save the settings to an appropriate destination.
4. Use this file and the option tables in this section to verify whether the options you used in your old project are still available or needed. Also check whether you need to use any of the new options.

For information about where to set the equivalent options in the IAR Embedded Workbench IDE, see the *78K IAR C/C++ Compiler Reference Guide*.

Removed options

The following table shows the command line options that have been removed:

Old option	Description
`-C`	Nested comments
`-d`	Static locals
`-Es`	Switch statement code inline
`-Eu`	Disables word alignment for data segments
`-Err`	No use of register variables
`-F`	Form-feed in list file after each function
`-G`	Open standard input as source; replaced by - (dash) as source file name in version 4.x
`-g`	Global strict type checking; in version 4.x, global strict type checking is always enabled
`-g0`	No type information in object code
`-h`	Disable assignment compatibility attribute test
`-i`	Add #include file text
`-K`	'`//`' comments; in version 4.x, '`//`' comments are allowed unless the option `--strict_ansi` is used
`-P`	Generate promable code
`-p`*nn*	Lines/page
`-R`	Code segment name
`-T`	Active lines only
`-t`*n*	Tab spacing
`-U`*symb*	Undefined preprocessor symbol
`-X`	Explain C declarations
`-x[DFT2]`	Cross-reference
`-y`	Writable strings

Table 2: Version 3.x compiler options not available in version 4.x

Identical options

The following table shows the command line options that are *identical* in version 3.x and version 4.x:

Option	Comment
`-D`*`symb`*`=`*`value`*	Define symbols
`-e`	Language extensions
`-f` *`filename`*	Extend the command line
`-I`	Include paths (The syntax is more free in ICC78K version 4.x)
`-o` *`filename`*	Set object filename
`-s[0–9]`	Optimize for speed
`-z[0–9]`	Optimize for size

Table 3: Compiler options identical in both compiler versions

Note: For the optimization options (`-s` and `-z`), only levels 2, 3, 6, and 9 are available in version 4.x.

Renamed or modified options

The following version 3.x command line options have been *renamed* and/or *modified*:

Old option	New option	Description
`-A` `-a` *`filename`*	`-la .` `-la` *`filename`*	Assembler output; see *Filenames*, page 11
`-b`	`--library_module`	Makes an object a library module
`-c`	`--char_is_signed`	`'char'` is `'`signed `char'`
`-gA`	`--strict_ansi`	Flags old-style functions
`-H`*`name`*	`--module_name=`*`name`*	Sets object module name
`-L[`*`prefix`*`]`, `-l` *`filename`*	`-l[a|A|b|B|c|C|D][N][H]` `{`*`filename`*`|`*`directory`*`}`	Generates list file; the modifiers specify the type of list file to create
`-ms`	`--code_model standard`	Model for non-banked function calls; library functions are called with the `CALL` instruction
`-mS`	`--code_model standard` `--generate_callt_runtime_` `library_calls`	Model for non-banked function calls; library functions are called with the `CALLT` instruction
`-mb`	`--code_model banked`	Model for banked function calls; library functions are called with the `CALL` instruction

Table 4: Renamed or modified options

Old option	New option	Description
-mB	--code_model banked --generate_callt_runtime_library_calls	Model for banked function calls; library functions are called with the CALLT instruction
-N*prefix*, -n *filename*	--preprocess=[c][n][l] *filename*	Preprocessor output
-q	-lA . -lC .	Inserts mnemonics; list file syntax has changed
-r[012][i][n][r][e]	-r --debug	Generates debug information; the modifiers have been removed
-S	--silent	Sets silent operation
-u	--disable_data_alignment	Disables data alignment of data objects
-v[0|1|2]	--core=[78k0_basic|78k0|78k0s]	Specifies the microcontroller core
-W{rs}	--workseg_area{=rs}	Specifies the space reserved in the saddr area for the WRKSEG segment.
-w	--no_warnings	Disables warnings

Table 4: Renamed or modified options (Continued)

FILENAMES

In version 3.x, file references can be made in either of the following ways:

- With a specific filename, and in some cases with a default extension added, using a command line option such as -a *filename* (assembler output to named file).
- With a prefix string added to the default name, using a command line option such as -A[*prefix*] (assembler output to prefixed filename).

In version 4.x, a file reference is always regarded as a *file path* that can be a directory which the compiler will check and then add a default filename to, or a *filename*.

The following table shows some examples where it is assumed that the source file is named test.c, myfile is *not* a directory and mydir is a directory:

Old command	New command	Result
-l myfile	-l myfile	myfile.lst
-Lmyfile	-l myfiletest	myfiletest.lst
-L	-l .	test.lst
-Lmydir/	-l mydir	mydir/test.lst

Table 5: Specifying filename and directory in version 3.x and version 4.x

LIST FILES

In version 3.x, only one C list file and one assembler list file can be produced; in version 4.x there is no upper limit on the number of list files that can be generated. The new command line option `-l[a|A|b|B|c|C|D][N][H] {filename|directory}` is used for specifying the behavior of each list file.

Extended keywords

The set of extended keywords has changed in version 4.x. Some keywords have been added, some keywords have been removed, and for some keywords the syntax has changed. In addition, memory attributes have a different interpretation if used in combination with `typedef`.

In version 4.x, all extended keywords except `asm` start with two underscores, for example `__no_init`.

The following table lists the old keywords, their new equivalents, and completely new keywords:

Old keyword/construction	New keyword/construction
`_ASM` (this was an intrinsic function in version 3.x)	`asm`
`banked`	`__banked`
`bit`	Bit variables are now supported by using 1-bit `char` bitfields
`callf`	`__callf`
`callt`	`__callt`
`interrupt`	`__interrupt`
`monitor`	`__monitor`
`near`	`__near`
`non_banked`	`__non_banked`
`no_init`	`__no_init`
`no_save`	`__no_save`
`saddr`	`__saddr`
`shortad`	`__saddr`
`sfr`	SFRs are declared using absolute declarations with the `@` or `#pragma location` syntax together with the keywords `__sfr __no_init volatile char`

Table 6: Old and new extended keywords

Old keyword/construction	New keyword/construction
`sfrp`	Declaration of SFRs is done by using absolute declarations with the `@` or `#pragma location` syntax together with the keywords `__sfr __no_init volatile short`
`sfr...__IO_NB`	`__no_bit_access`
`sfr...__IO_RO`	`const`
`sfr...__IO_WO`	The SFR write only access is no longer available
`using[n]`	The bank number *n* has to be declared using `#pragma bank=n`

Table 6: Old and new extended keywords (Continued)

To simplify the migration, the include file `migration.h` is delivered with version 4.x. This include file maps the old keywords with their new counterparts, if possible. For example: `#define near __near`

Whenever possible, the include file also maps old and new intrinsic functions.

For detailed information about the extended keywords available in version 4.x, see the *78K IAR C/C++ Compiler Reference Guide*.

SFR

In version 3.x of the compiler, the keywords `sfr` and `sfrp` denote an object of byte or word size residing in the Special Function Register (SFR) memory area for the chip, and having a `volatile` type. The SFR is always located at an absolute address. For example:

```
sfr P0=FF00;
```

In version 4.x, the keywords `sfr` and `sfrp` are not available. Instead is possible to:

- Place any object into any memory, by using a memory attribute; for example:
  `__near int b;`
- Locate any object at an absolute address by using the `#pragma location` directive or by using the locator operator `@`; for example:
  `long P0 @ FF00;`
- Use the `volatile` attribute on any type, for example:
  `volatile char P0 @ FF00;`

BIT VARIABLES

A bit variable in version 3.x is a `volatile` boolean variable that can have an absolute bit address, be co-located with an SFR or be a relocatable object, like ordinary variables. For example:

```
bit a = 87; /* at bit-address 87 (version 3.x) */
bit p0_bit = P0.5; /* bit 5 of P0 (version 3.x) */
```

```
bit r; /* relocatable bit (version 3.x) */
```

Version 4.x uses bitfields of width 1 to implement bit variables. The extended language feature *anonymous structures* allows the bits, which are structure members, to be used as if they were variables in the enclosing scope. The keyword `bit` is not available in version 4.x.

The following example shows an anonymous structure in version 4.x:

```
/* anonymous structure */
struct {
  char b0:1, b1:1, b2:1, :5, b7:1;
};
char foo() { return b7; }
void bar() { b0 = 1; }
```

Anonymous unions are used for locating an SFR and a bitfield at the same address. The declaration of `P0` (address `0xFF00`) and the bits of `P0` are combined in the following way:

```
/* anonymous union */
extern __sfr __no_init volatile union
{
  unsigned char P0;
  struct
  {
    unsigned char no0:1;
    unsigned char no1:1;
    unsigned char no2:1;
    unsigned char no3:1;
    unsigned char no4:1;
    unsigned char no5:1;
    unsigned char no6:1;
    unsigned char no7:1;
  }P0_bit;
} @ 0xFF00;
```

The version 4.x notation is not as brief as the one used in version 3.x. It is, on the other hand, more flexible. Bitfields can have any width (not only 1), can be located in any memory and are not necessarily `volatile`. The same (maximal) packing as in version 3.x can be achieved by placing all bits in the same anonymous structure.

STORAGE MODIFIERS

Both version 3.x and version 4.x allow keywords that specify memory location of an object—memory attributes. Each of these attributes can be used either as a placement attribute for an object, or as a pointer type attribute denoting a pointer that can point to the specified memory.

When the attributes are used directly in the source code, they behave in a similar way in both compiler versions. However, the usage of memory attributes in combination with the keyword typedef is more strict in version 4.x than in version 3.x.

Version 3.x behaves unexpectedly in some cases:

```
typedef int __near NINT;
NINT a,b;
NINT __saddr c;   /* Illegal */
NINT *p;          /* p stored in near memory, points to default
                     memory type */
```

The first variable declaration works as expected, that is `a` and `b` are located in near memory. However, the declaration of `c` is illegal.

In the last declaration, the `__near` keyword of the type definition affects the location of the pointer variable `p`, not the pointer type. The pointer type is default.

The corresponding example for version 4.x is:

```
typedef int __near NINT;
NINT a,b;
NINT __saddr c;   /* c stored in saddr memory; override attribute
                     in typedef */
NINT *p;          /* p stored in default memory, points to
                     near memory */
```

The declaration of `c` and `p` differ. The `__saddr` keyword in the declaration of `c` will always compile. It overrides the keyword of the `typedef`. In the last declaration the `__near` keyword of the `typedef` affects the type of the pointer. It is thus a `__near` pointer to `int`. However, the location of the variable `p` is not affected.

INTERRUPT FUNCTIONS AND VECTORS

The syntax for defining interrupt functions has changed from version 3.x.

Old syntax

The syntax when defining interrupt functions using version 3.x:

```
interrupt [vector] using [bankno] void function_name(void);
```

where *`vector`* is the vector offset in the vector table and *`bankno`* is the register bank to be used.

New syntax

The syntax when defining interrupt functions using version 4.x:

```
#pragma bank=bankno
#pragma vector=vector
__interrupt void function_name(void);
```

where *vector* is the vector offset in the vector table and *bankno* is the register bank to be used.

For further details of the new pragma directives, see the *78K IAR C/C++ Compiler Reference Guide*.

ABSOLUTE LOCATED VARIABLES

In version 3.x the syntax was:

```
sfr PORT = 0xFF10;
```

The extended keyword `sfr` is exchanged with the keyword `__sfr` in version 4.x. Note that the new keyword has a different syntax.

In version 4.x you can:

- Locate any object at an absolute address by using the `#pragma location` directive, or by using the locator operator `@`, for example:
  `__no_init long PORT @ 100;`
- Use the `volatile` attribute on any type, for example:
  `__sfr __no_init volatile unsigned char PORT @ 0xFF10;`

For further details about how to locate variables, see the *78K IAR C/C++ Compiler Reference Guide*.

Pragma directives

Version 3.x and version 4.x have different sets of pragma directives for specifying attributes, and they also behave differently:

- In version 3.x, `#pragma memory` specifies the default location of data objects, and `#pragma function` specifies the default location of functions. They change the default attribute to use for declared objects; they do not have an effect on pointer types.
- In version 4.x, the `#pragma type_attribute` and `#pragma object_attribute` directives only change the next declared object or `typedef`.

See the *78K IAR C/C++ Compiler Reference Guide* for information about the pragma directives available in version 4.x.

The following pragma directives have been removed:

- `codeseg`
- `function`
- `memory`
- `warnings`

They are recognized and will give a diagnostic message but will not work in version 4.x.

Note: Instead of the `#pragma codeseg` directive, the `#pragma location` directive or the `@` operator can be used for specifying an absolute location.

The following table shows the mapping of pragma directives:

Old directive	New pragma directive
`#pragma function=interrupt`	`#pragma type_attribute=__interrupt`
`#pragma function=monitor`	`#pragma type_attribute=__monitor`
`#pragma memory=constseg`	`#pragma constseg, #pragma location`
`#pragma memory=dataseg`	`#pragma dataseg, #pragma location`
`#message`	`#pragma message`

Table 7: Old and new pragma directives

It is important to note that the new directives `#pragma type_attribute`, `#pragma object_attribute`, and `#pragma vector` affect only the *first* of the declarations that follow after the directive. In the following example, `x` is affected, but `z` and `y` are not affected by the directive:

```
#pragma object_attribute=__no_init
int x,z;
int y;
```

Specific segment placement

In version 3.x, the `#pragma memory` directive supports a syntax that enables subsequent data objects that match certain criteria to end up in a specified segment. Each object found after the invocation of a segment placement directive will be placed in the segment, provided that it does not have a memory attribute placement, and that it has the correct constant attribute. For `constseg`, it must be a constant, while for `dataseg`, it cannot be declared `const`.

In version 4.x, the directive `#pragma location` and the `@` operator are available for this purpose.

Intrinsic functions

Version 4.x has a new naming convention for intrinsic functions, as well as a large number of additional functions.

The old intrinsic functions `_OPC`, `_args$` and `_argt$` available in version 3.x are removed and cannot be used in version 4.x. However, except for these three functions, all intrinsic functions available in version 3.x can be used also in version 4.x.

To use the old intrinsic functions, include the files `migration.h` and `intrinsics.h`. To use only the new intrinsic functions, include only the file `intrinsics.h`.

Note: The compiler option `-e` (**Enable IAR C extended language**) must be selected.

The following table lists the old intrinsic functions and their new equivalents, as well as the new intrinsic functions:

Old intrinsic function	New intrinsic function	Description
`_args$`	None	Returns an array of the parameters to a function.
`_argt$`	None	Returns the type of a parameter.
`_ASM`	`asm` (this is an extended keyword, and not an intrinsic function, in version 4.x)	Inserts an assembler statement.
`_BRK`	`__break`	Inserts a `BRK` instruction.
`_DI`	`__disable_interrupt`	Disables interrupts by inserting the `DI` instruction.
`_EI`	`__enable_interrupt`	Enables interrupts by inserting the `EI` instruction.
–	`__get_interrupt_state`	Returns the current interrupt state.
`_HALT`	`__halt`	Inserts a `halt` / `nop` instruction pair.
`_NOP`	`__no_operation`	Generates a `NOP` instruction.
`_OPC`	None	Inserts a byte constant.
–	`__segment_begin`	Returns the start address of a segment.
–	`__segment_end`	Returns the end address of a segment.
–	`__set_interrupt_state`	Sets the current interrupt state.
`_STOP`	`__stop`	Inserts a stop / no-operation instruction pair.

Table 8: Old and new intrinsic functions

See the *78K IAR C/C++ Compiler Reference Guide* for further information about the intrinsic functions available in version 4.x.

Segments

The segment naming convention has changed since the version 3.x. Some of the old segments have disappeared, and some new ones have been introduced.

For details of the new segments, their names, and how they are used, *78K IAR C/C++ Compiler Reference Guide*.

This table lists the old segment names, their counterparts in version 4.x, and additional segments:

Old segment	New segment
BITVARS	–
CCSTR [1]	–
CDATA0, CDATA1	SADDR_ID
CDATA2	NEAR_ID
CODE	CODE, BCODE
CONST	CONST
CSTACK	CSTACK
CSTR	CONST
ECSTR [1]	–
FCODE	FCODE
FLIST, IFLIST	CLTVEC
IDATA0, IDATA1	SADDR_I
IDATA2	NEAR_I
INTVEC	INTVEC
NO_INIT	NEAR_N, SADDR_N
RCODE	RCODE
UDATA0, UDATA1	SADDR_Z
UDATA2	NEAR_Z
WRKSEG	WRKSEG
–	NEAR_AC [2]
–	NEAR_AN [2]
–	DIFUNC
–	HEAP
–	SADDR_AC [2]
–	SADDR_AN [2]

Table 9: Old and new segments

Old segment	New segment
–	SWITCH

Table 9: Old and new segments (Continued)

1) Version 4.x does not support placing strings in writable memory. For this reason, the old segments used for this task have no counterparts in version 4.x.
2) Segments ending in _AN and _AC contain data located at absolute addresses, and should not be included in the linker command file.

LINKER COMMAND FILE CONSIDERATIONS

If you have created your own customized linker command file, compare this file with the original file in the old installation and make the required changes related to segment names in a copy of the corresponding file in the new installation.

New linker segment control directive -P supersedes -b

If you earlier used the linker segment control directive `-b` for locating your banked code in memory, be aware that this directive is now obsolete and has been superseded by the new directive `-P`. If you have an old linker command file (`.xcl`) that uses the `-b` directive, you should replace those instances with the `-P` directive in a way that has an equivalent effect.

The `-b` linker directive is intended exclusively for placement of banked segments and uses a special syntax to specify the ranges where the segments are placed. It has a limited support for address translation. In addition to this, range errors generated by banked code are suppressed because earlier versions of the compiler were unable to generate correct range checks for banked code.

The `-P` linker directive, on the other hand, is used for packed segment placement, and uses the linker directive `-M` to perform address translation.

Code segments generated by the current compiler sometimes rely on the linker to place them in certain ways. Because segment placement using `-b` does not use the linker this way, it is unsafe to use `-b` for placing banked code generated by the current compiler.

It is important to understand how the `-b` directive works, to express the same thing with a `-P` command. You must also note that the current compiler might generate code in a different way than earlier versions and that the name and size of segments have changed. In addition, there might also be dependencies between the segments that were not there before. Therefore, you should carefully consider whether you really want an exact translation.

How -b works

The basic `-b` command looks like this:

`-b(CODE)A,B,C,D=`*`START`*`,`*`LENGTH`*`,`*`INCREMENT`*`,`*`COUNT`*

This command allocates the segments A, B, C, and D. The governing parameters are:

Parameter	Description
START	The start address of the first bank.
LENGTH	The length (size) in bytes of each bank.
INCREMENT	The address interval between each bank. Each subsequent bank after the first starts at the address *START* + *x* * *INCREMENT* where x is the bank number.
COUNT	The number of banks available in this placement command, numbered 0 to *COUNT*-1

Table 10: Parameters to -b

Note: *START*, *LENGTH*, and *INCREMENT* are all hexadecimal numbers, *COUNT* is decimal.

Placement starts with the first segment, A. The segment parts of A are placed contiguously in bank 0 until all parts have been placed or there is not enough space left in bank 0. When all parts of segment A have been placed, the placement of the next segment (B) starts in the same bank (bank 0). If there is not enough space in bank 0 to place all parts of segment A, the placement of A continues in the next bank (bank 1).

Once the segment placement has left a bank, nothing more will be allocated to it, even if there actually is space available there.

How -P works

Packed segment placement looks like this:

```
-P(CODE)A,B,C,D=[RANGE_START-RANGE_END]*NUMBER_OF_RANGES+INCREMENT
```

This command places the segments A, B, C, and D. There are a total of *NUMBER_OF_RANGES* ranges, numbered 0 to *NUMBER_OF_RANGES*-1. Each range begins at address *RANGE_START* + x * *INCREMENT* where x is the range number.

The segments can be placed anywhere in any of these ranges. This allows for a more dense segment placement; if there is some available space in a range that was too small for a certain segment part, that range can be used by the linker for another segment part later on.

How the -b and -P directives correspond to each other

Regard this example:

```
-b(CODE)A,B,C,D=4000,1000,10000,2
```

This command has two available ranges (*COUNT* = 2).

Bank 0 starts at address 0x4000 (*START* = 4000) and is 0x1000 (*LENGTH* = 1000) bytes long (ending at address 0x4FFF). Bank 1 starts at address 0x14000 (*INCREMENT* = 10000) and ends at 0x14FFF. Expressed as a range in XLINK this would be 4000-4FFF,14000-14FFF.

In other words, this command places its segments into two banks, one at 4000–4FFF and another at 14000–14FFF. The equivalent using the packed segment placement -P directive would be:

```
-P(CODE)A,B,C,D=4000-4FFF,14000-14FFF
```

Another way of achieving the same thing is to use:

```
-P(CODE)A,B,C,D=[4000-4FFF]*2+10000
```

or

```
-P(CODE)A,B,C,D=[4000:+1000]*2+10000
```

The [4000-4FFF]*2+10000 style will be used in the rest of this section.

More examples

Here is another example:

```
-b(CODE)A,B,C,D=C000,4000,10000,5
```

There are a total of five banks, each one 0x4000 bytes long, and the first one starts at address 0xC000. The distance between the banks is 0x10000 bytes. The possible address ranges are: C000–FFFF, 1C000–1FFFF, 2C000–2FFFF, 3C000–3FFFF, and 4C000–4FFFF. Using the -P directive, this can be expressed as:

```
-P(CODE)A,B,C,D=[C000-FFFF]*5+10000
```

An example with no maximum number of banks:

```
-b(CODE)A,B,C=4000,8000,10000
```

Three segments are placed. There is no limit on the number of banks (*COUNT* is missing), each bank is 0x4000 bytes long, and the first bank starts at the address 0x4000. The distance between the banks is 0x20000 bytes. Expressed as a packed segment placement, this could look like this:

```
-P(CODE)A,B,C=[4000-BFFF]*3+10000
```

Note: The -b directive allows open ranges (no maximum number of banks) while -P requires closed ranges (a fixed number of ranges). It is impossible to know in advance that three banks will be needed; the segments might require 1, 2, 20, or perhaps even more banks. The number of banks to use must be based on information about the sizes of the segments that are placed.

Colon-separated segment lists

The -b directive can be used with colon-separated lists of segments:

```
-b(CODE)A:B:C:D=START,LENGTH,INCREMENT,COUNT
```

This syntax (meaning that each segment will be placed in a new bank) has no direct -P equivalent; usually a single -b command with a colon-separated segment list must be replaced with several -P commands, one for each segment.

See this example:

```
-b(CODE)A:B:C:D=1000,8000,10000
```

This command locates the segments A, B, C, and D. There is no limit to the number of banks, each bank is 0x8000 bytes long, and the first bank starts at the address 0x1000. The distance between the banks is 0x10000 bytes and each segment must be placed in its own bank. Using -P, this could look like this:

```
-P(CODE)A=1000-8FFF
-P(CODE)B=11000-18FFF
-P(CODE)C=21000-28FFF
-P(CODE)D=31000-38FFF
```

This assumes that each segment will fit in a single bank. If, for instance, segment B requires 3 banks and segment D requires 2 banks it could look like this:

```
-P(CODE)A=1000-8FFF
-P(CODE)B=[11000-18FFF]*3+10000
-P(CODE)C=[41000-48FFF]*2+10000
-P(CODE)D=61000-68FFF
```

Address translation using the -M directive

If the -b placement command you want to replace uses # (-b#, linear physical addresses) or @ (-b@, 64180-type physical addresses) you must also use the XLINK address translation directive -M to achieve the desired result.

See this example:

```
-b#(CODE)A,B,C=8000,4000,10000,2
```

The ranges are 8000–BFFF and 18000–1BFFF, so the -P equivalent might be:

```
-P(CODE)BANKED=[8000-BFFF]*2+10000
```

The linear physical addresses directive used with the -b directive specifies that the banks are placed contiguously in memory. The addresses in this case would be 8000–BFFF, C000–FFFF. Using the -M directive, this can be expressed as:

```
-M(CODE)[8000-BFFF]*2+10000=8000
```

Note: We do not have to map the address to the address `0x8000`, we could use any address. `-M` is a more powerful replacement for the limited address translation offered by `-b#` and `-b@`.

For details of the `-P` directive, see the *IAR Linker and Library Tools Reference Guide*.

Other changes

This section describes changes related to:

- Object file format
- Predefined symbols
- Nested comments
- `Sizeof` in preprocessor directives
- Floating-point arithmetics.

OBJECT FILE FORMAT

In version 3.x, two types of source references can be generated in the object file. When the command line option `-r` is used, the source statements are being referred to. When the command line option `-re` is used, the actual source code is embedded in the object format.

In version 4.x, when the command line option `-r` or `--debug` is used, source file references are always generated. Embedding of the source code is not supported.

PREDEFINED SYMBOLS

All predefined symbols supported in version 3.x are also supported in version 4.x. Version 4.x, however, has additional ones.

The predefined symbol `__IAR_SYSTEMS_ICC` is provided only for compatibility with version 3.x. Version 4.x also has the `__IAR_SYSTEMS_ICC__` symbol.

See the *78K IAR C/C++ Compiler Reference Guide* for information about the predefined symbols available in version 4.x.

NESTED COMMENTS

In the old version, nested comments are allowed if the option `-C` is used. In version 4.x, nested comments are never allowed. For example, if a comment was used for removing a statement as in the following example, it would not have the desired effect.

```
/*
/* x is a counter */
int x = 0;
*/
```

The variable `x` will still be defined, there will be a warning where the inner comment begins, and there will be an error where the outer comment ends.

```
  /* x is a counter */
  ^
"c:\bar.c",2  Warning[Pe009]: nested comment is not allowed

  */
   ^
"c:\bar.c",4  Error[Pe040]: expected an identifier
```

The solution is to use `#if 0` to "hide" portions of the source code when compiling:

```
#if 0
/* x is a counter */
int x = 0;
#endif
```

Note: `#if` statements may be nested.

SIZEOF IN PREPROCESSOR DIRECTIVES

In version 3.x, `sizeof` could be used in `#if` directives, for example:

```
#if sizeof(int)==2
int i = 0;
#endif
```

In version 4.x, `sizeof` is not allowed in `#if` directives. The following error message will be produced:

```
  #if sizeof(int)==2
      ^
"c:\bar.c",1  Error[Pe059]: function call is not allowed in a
constant expression.
```

Macros can be used instead, for example `SIZEOF_INT`. Macros can be defined using the `-D` option, or a `#define` in the source code:

```
#if SIZEOF_INT==2
int i = 0;
#endif
```

To find the size of a predefined data type, see *78K IAR C/C++ Compiler Reference Guide*.

Complex data types may be computed using one of two methods:

- Write a small program and run it in the simulator, with terminal I/O.

```
#include <stdio.h>
struct s { char c; int a; };

void main(void)
{
  printf("sizeof(struct s)=%d \n", sizeof(struct s));
}
```

- Write a small program, compile it with the option `-la .` to get an assembler listing in the current directory, and look for the definition of the constant `x`.

```
struct s { char c; int a; };
const int x = sizeof(struct s);
```