

# Migration guide

## Migrating from Keil $\mu$ Vision<sup>®</sup> for 8051 to IAR Embedded Workbench<sup>®</sup> for 8051

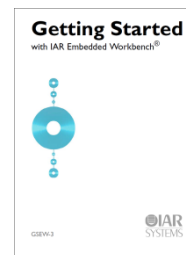
Use this guide as a guideline when converting project files from the  $\mu$ Vision IDE and source code written for Keil toolchains for 8051 to IAR Embedded Workbench for 8051.

	Product	Version number
Migrating from	Keil $\mu$ Vision IDE (C51 compiler)	V4.x
Migrating to	IAR Embedded Workbench for 8051	V9.30 and newer

### Migration overview

Migration of an existing project from Keil  $\mu$ Vision requires that you collect information about your current project and then apply this information to the new IAR Embedded Workbench project. In addition, you need to make some changes in the actual source code. The information in this guide simplifies this process.

**Note:** Basic introduction to IAR Embedded Workbench and how to work in the IDE can be found in the guide [Getting Started with IAR Embedded Workbench](#) available in the Information Center and in the tutorials.



### Project conversion

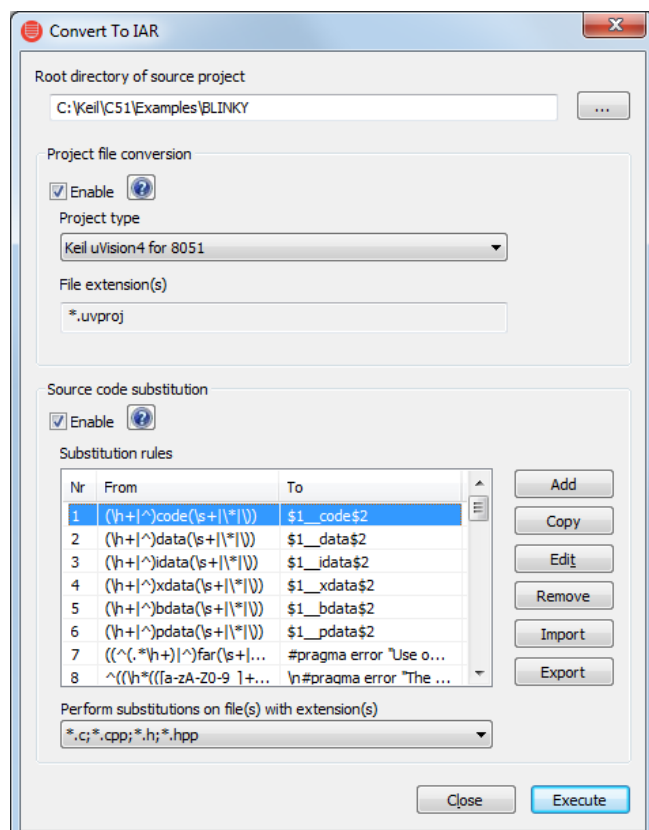
To migrate existing Keil  $\mu$ Vision applications to IAR Embedded Workbench there is a tool called **Convert To IAR**. This is a GUI application included with IAR Embedded Workbench, available via the **Tools** menu.

The **Convert To IAR** tool converts  $\mu$ Vision 4 project files into IAR Embedded Workbench project files without changing the original file. Information about source files, include paths, defined symbols, and build configuration is transferred. As an option, also source code substitutions are performed and you can add your own substitution rules including support for regular expressions. There are a number of pre-defined substitution rules.

#### Procedure

1. Start IAR Embedded Workbench.
2. Start **Convert To IAR** available in the **Tools** menu.
3. Navigate to the  $\mu$ Vision project to convert by clicking the browse button.
4. Click the **Execute** button and a new IAR Embedded Workbench project file will be created.
5. Add the new project to a workspace by choosing **Add Existing Project** in the **Project** menu.
6. Set the relevant project options by choosing **Options** in the **Project** menu.

Hint: Open the original project in  $\mu$ Vision, walk through the options and set the corresponding options in IAR Embedded Workbench as suggested in the section *Important tool settings* below.



## Basic code differences

The following sections show some of the basic differences between code written for the Keil toolchain and IAR Embedded Workbench that you should handle before building your converted project.

### Initialization code

In IAR Embedded Workbench, initialization code is primarily located in the file `cstartup.s51`.

This file contains system startup code executed after reset, but before the `main()` function is called. Data/segment initialization, stack pointer initialization and other things are performed here. This code is part of the runtime library but can be overridden by including a copy of this assembler file in your project. You find the file in the folder `8051\src\lib` in the IAR Embedded Workbench installation.

The function `int __low_level_init(void)` is called from `cstartup.s51`. Its purpose is to perform any hardware initialization required before segment initialization and calling `main()`. You may include your own version of this function in your project by adding a copy of the file `low_level_init.c`, located in the folder `8051\src\lib`, and edit it according to you needs.

### Special Function Registers

Each device supported by IAR Embedded Workbench has its own header file that contains variable definitions to access the SFRs of the device, both for C/C++ and assembler. The naming convention for these header files is `iodevice-name.h` and they are located in `8051\inc`.

Example: `ioEFM8SB20F64G.h`

The names of SFR variables might sometimes differ from the names used in the Keil toolchain. SFR bit access differs because IAR Embedded Workbench does not support the `sbit` keyword which is commonly used in the Keil toolchain for the purpose of defining variables to access specific SFR bits. Instead, a C `struct` where each member represents one or more bits is used for this purpose. This `struct` is named `SFR-name_bit`, where `SFR-name` is the name of the corresponding SFR variable represented as a byte.

Sometimes several SFR bits, which are semantically related within the same SFR, are represented by separate `sbit` variables in the Keil toolchain while they can be combined into a single bit-field in IAR Embedded Workbench. This can make it necessary to rewrite source code which manipulates such bits or define your own variables to access the bits separately.

The `__sfr` memory type attribute keyword is used in IAR Embedded Workbench to define/declare SFR variables, for example as follows:

```
__sfr __no_init volatile unsigned char TL0 @ 0x8A;
```

To define a variable which represents an SFR which consists of two bytes there is no special keyword like `sfr16` which is used in the Keil toolchain. The same keyword is used but the variable has the type `unsigned short` instead of `unsigned char`:

```
__sfr __no_init volatile unsigned short ADC0 @ 0xBD;
```

The same restrictions as in the Keil toolchain apply when using this type of variable, the low byte should immediately precede the high byte address wise and the low byte is the address of the variable. There is one difference though, when writing to the variable Keil writes the most significant byte first and the least significant byte last while IAR Embedded Workbench writes them in the opposite order. Generally, the writing order of SFRs can affect the hardware behavior so care must be taken when utilizing multi-byte SFRs.

### Interrupt Service Routines

The `interrupt` keyword is used in the Keil toolchain when defining an interrupt function (ISR). In IAR Embedded Workbench, the extended keyword `__interrupt` is used for the same purpose. The `interrupt` keyword has a numeric parameter which the Keil compiler automatically translates into an interrupt vector number. IAR Embedded Workbench requires the application developer to specify the vector number directly using the `vector` pragma directive. For convenience, the device-specific header file containing the SFR access variables also contain defined names for the interrupt vectors. Here is an example comparing the syntax used in the two toolchains:

## Migrating from Keil µVision for 8051 to IAR Embedded Workbench for 8051

Keil toolchain	IAR Embedded Workbench
<pre>void timer0_isr(void) interrupt 1 {     ... }</pre>	<pre>#pragma vector=0x0B __interrupt void timer0_isr(void) {     ... }</pre>

It is also possible to specify a register bank which will be used by an interrupt function. In the Keil toolchain, this can be done for any function, but in IAR Embedded Workbench it is only applicable to interrupt functions. The `using` keyword is used in the Keil toolchain whereas the `register_bank` pragma directive is used in IAR Embedded Workbench. Here is an example:

Keil toolchain	IAR Embedded Workbench
<pre>void timer0_isr(void) interrupt 1 using 2 {     ... }</pre>	<pre>#pragma register_bank=2 #pragma vector=0x0B __interrupt void timer0_isr(void) {     ... }</pre>

### Inline assembler

Inline assembler is handled somewhat differently in the Keil toolchain than in IAR Embedded Workbench. The Keil toolchain provides mechanisms to insert assembler code only if configured to generate a separate assembler source file (`.src`) which is then assembled to object code. IAR Embedded Workbench integrates the assembler code directly into the object code when compiling, but of course it is possible to generate a separate assembler source file (`.s51`) if desired.

The mechanism used in the Keil toolchain are the pragma directives `ASM` and `ENDASM` (or the alternative keyword `__asm`). There are no corresponding pragmas in IAR Embedded Workbench, the `asm` keyword (or its alias `__asm`) is used instead. Note that `__asm` in the Keil toolchain does not have the same syntax or semantics as `__asm` in IAR Embedded Workbench. Here is an example of two equivalent code sequences:

Keil toolchain	IAR Embedded Workbench
<pre>#pragma ASM     MOV A, #33H     MOV R1, #22H     ADD A, R1     JMP \$ #pragma ENDASM</pre>	<pre>__asm("MOV A, #0x33\n"       "MOV R1, #0x22\n"       "ADD A, R1\n"       "JMP \$");</pre>

### Managing memory

The *memory models* used in the Keil toolchain – which can be applied to functions, constants and variables – correspond to the concepts of *code model* and *data model* in IAR Embedded Workbench. The code and data models are set per project and specify the default storage model for functions and data respectively. The code model can be overridden for individual functions using *function memory attribute* extended keywords in the source code. The data model can be overridden for individual data objects or pointers by specifying *data memory attribute* extended keywords in the source code. The data memory attributes correspond to *memory type specifiers* in the Keil toolchain.

Data with an integer type represented by more than one byte is stored as big endian in the Keil toolchain. In IAR Embedded Workbench such data is stored as little endian. This fact might require source code to be rewritten to work as expected.

Many library functions declared in `string.h` (such as `memcpy`, `memcmp`, `strcat` etc.) cannot handle parameters which are located in different memory types. This might cause unexpected results.

### Bit variables

Declaring a single-bit variable (which will be located in the bit-addressable area of the internal RAM) is done in the Keil toolchain using the type *bit*. In IAR Embedded Workbench, this is accomplished by declaring a variable which has the type *bool* in combination with the `__bit` data memory attribute. A declaration using the `__bit` attribute has limitations; it must be located outside the function scope, and the object attribute `__no_init` must also be used which means that such a variable cannot be initialized at the same time it is declared. If, in code written for the Keil toolchain, a variable declared with the *bit* keyword resides within a function, a function argument list, or if a function returns a value of type *bit*, a supported type such as *bool* should be used in IAR Embedded Workbench.

## Migrating from Keil $\mu$ Vision for 8051 to IAR Embedded Workbench for 8051

Variables previously declared using the type *bit* have another type in IAR Embedded Workbench. Therefore they are no longer bit-oriented and references to such variables should be reviewed carefully because bit-oriented operators such as  $\sim$ ,  $\wedge$ ,  $\&$  and  $|$  will most probably not have the same effect after the change of type.

### Building your project

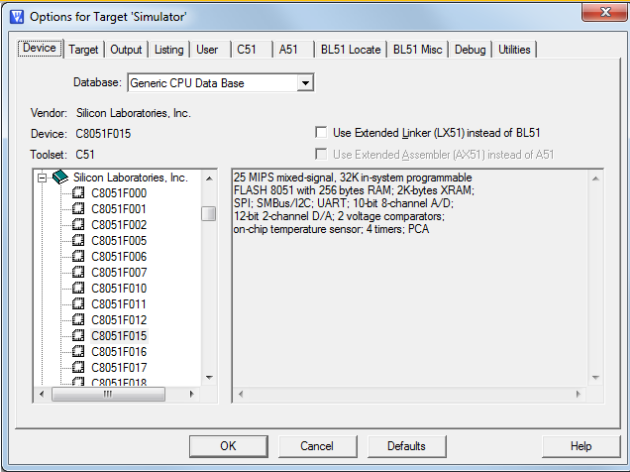
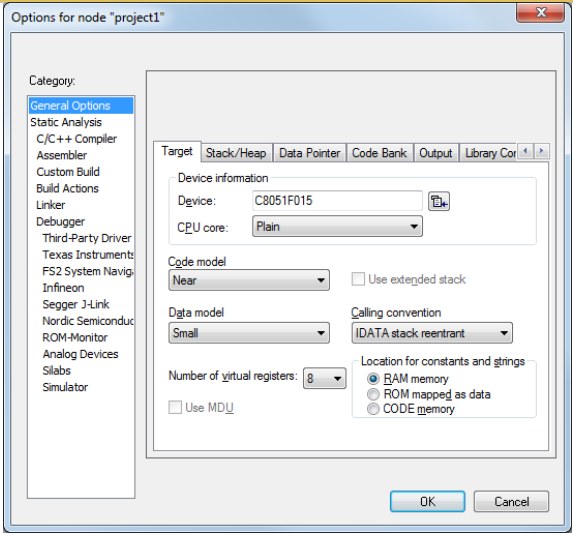
After successfully converting the Keil  $\mu$ Vision project and considered the basic code differences described above, you will still most likely need to fine-tune parts of the source code so that it follows the IAR Embedded Workbench syntax.

1. Select your device under **Project>Options>General Options**.
2. Choose **Project>Make**.
3. To find the different errors/warnings, press **F4** (Next Error/Tag).  
This will bring you to the location in the source code that generated this error/warning.
4. For each error/warning, modify the source code to match the IAR Embedded Workbench syntax.  
Note: You might have to consult the [IAR C/C++ Compiler User Guide](#) for this step.
5. After correcting one or more errors/warnings, repeat the procedure.

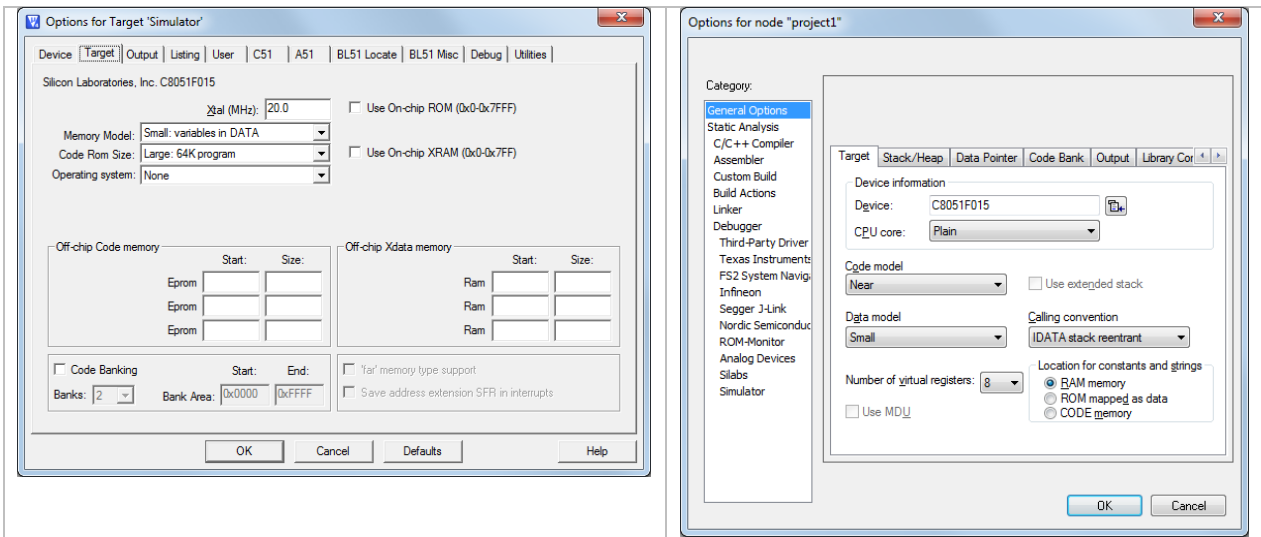
Note: It is always a good idea to start by correcting the first couple of errors/warnings in different source files because errors and warnings later in the source code might just be effects of faulty syntax at the beginning of the source code.

### Important tool settings

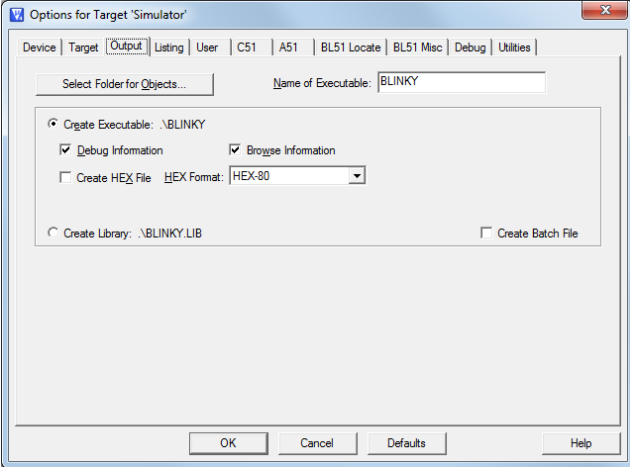
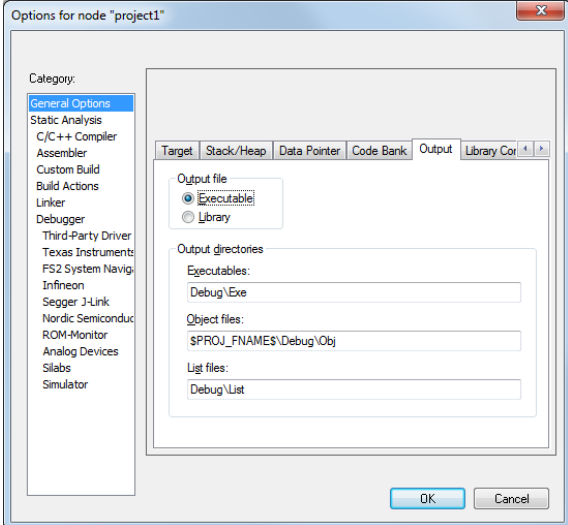
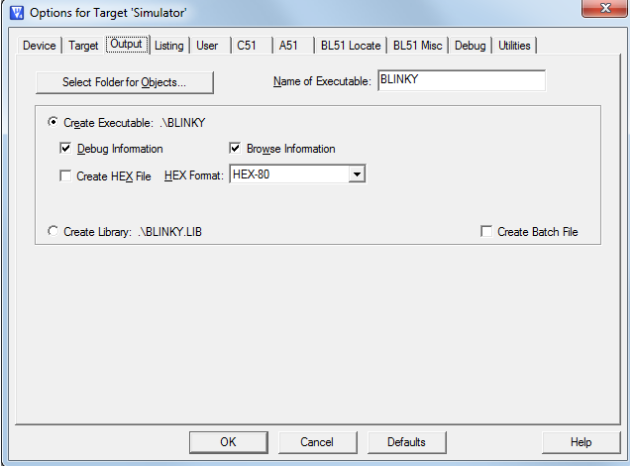
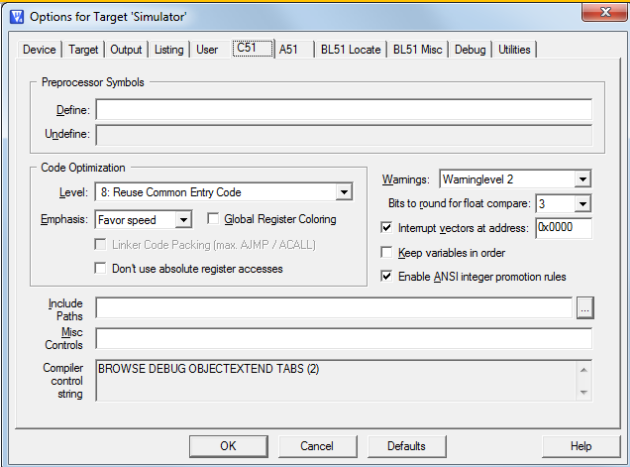
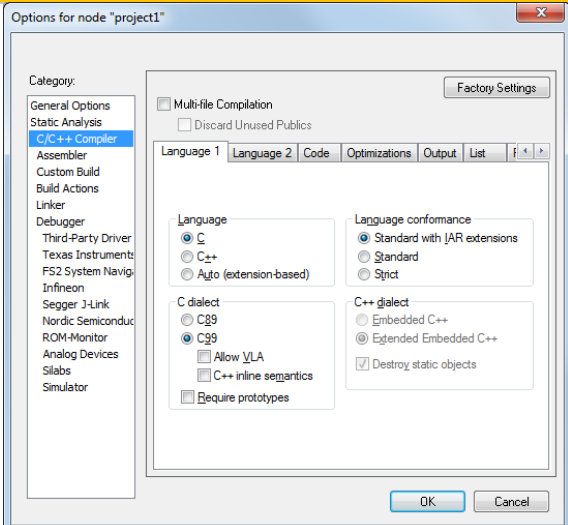
This is an overview of the most important tool settings. Note that many settings do not have a one-to-one mapping. An example is the memory segment configuration which is available in Keil  $\mu$ Vision through the Options dialog box. In IAR Embedded Workbench segment configuration is all done in the linker configuration file which can be pointed out in the **Options** dialog box.

Keil $\mu$ Vision	IAR Embedded Workbench
<b>Device selection</b> 	<b>Options for node "project1"</b> 
<b>Default memory model</b>	

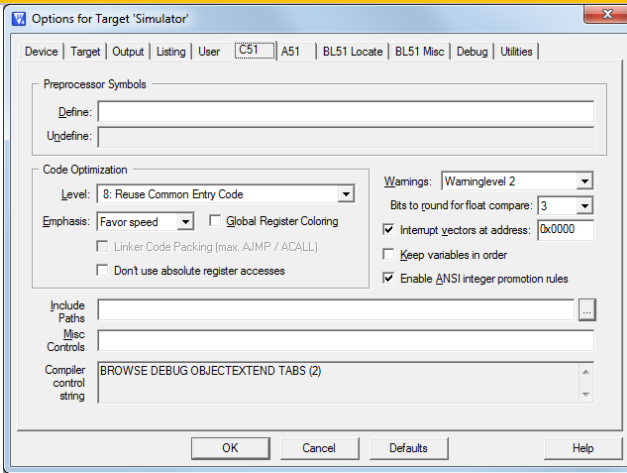
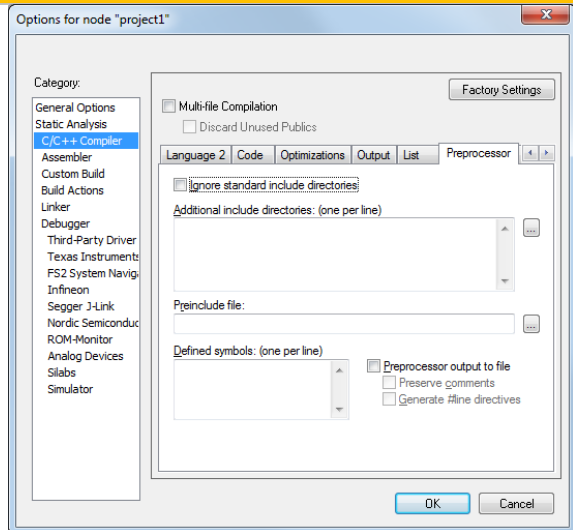
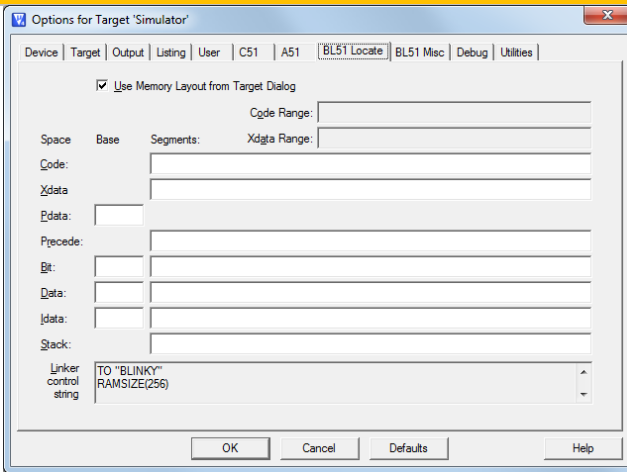
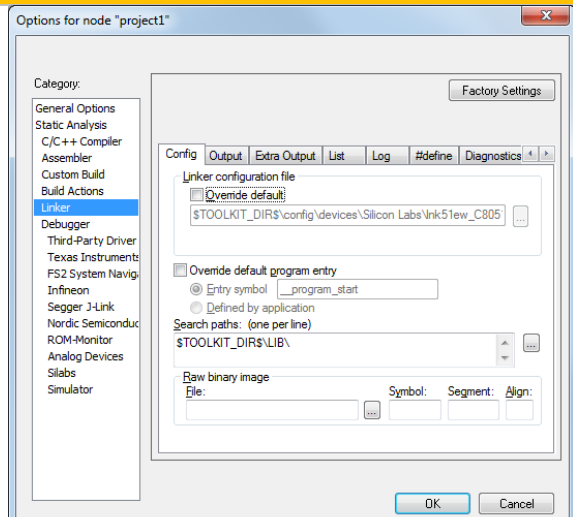
# Migrating from Keil µVision for 8051 to IAR Embedded Workbench for 8051



Migrating from Keil  $\mu$ Vision for 8051 to IAR Embedded Workbench for 8051

Keil $\mu$ Vision	IAR Embedded Workbench
Output type	Output type
	
Output format	Output format
	
Compiler options	Compiler options
	

# Migrating from Keil µVision for 8051 to IAR Embedded Workbench for 8051

Keil µVision	IAR Embedded Workbench
<b>Defined symbols and include directories</b>	
	
<b>Linker options</b>	
	

Note: We recommend that you verify all settings to make sure they match your project requirements.

IAR, IAR Systems, IAR Embedded Workbench, C-SPY, C-RUN, C-STAT, visualSTATE, Focus on Your Code, IAR KickStart Kit, IAR Experiment!, I-jet, I-jet Trace, I-scope, IAR Academy, IAR, and the logotype of IAR Systems are trademarks or registered trademarks owned by IAR Systems AB.

All information is subject to change without notice. IAR Systems assumes no responsibility for errors and shall not be liable for any damage or expenses.

© 2016 IAR Systems AB. Part number: EW8051\_MigratingFromKeil-2. Second edition: January 2016