AVR® IAR Embedded Workbench® IDE

Migration Guide

for Atmel® Corporation's

AVR® Microcontroller

EDITION NOTICE

First edition: May 2007

Part number: MAVR-2

This guide applies to version 4.x of AVR IAR Embedded Workbench®.

Migrating from version 3.x to version 4.x

This chapter gives hints for porting your application code and projects to the new version 4.x.

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

This chapter presents the major differences between the AVR IAR Embedded Workbench version 3.x and the AVR IAR Embedded Workbench version 4.x, and describes the migration considerations.

Note that if you are migrating from AVR IAR Embedded Workbench version 1.x (EWA90), you must first read the chapter *Migrating from EWA90 to EWAVR*.

IAR Embedded Workbench IDE

Upgrading to the new version of the IAR Embedded Workbench IDE should be a smooth process as the improvements do not affect the compatibility between the versions.

WORKSPACE AND PROJECTS

The workpaces and projects you have created with 3.x are compatible with 4.x.

C-SPY LAYOUT FILES

Due to a new improved window management system, the C-SPY layout files support in 3.x has been removed. Any custom-made `lew` files can safely be removed from your projects.

Runtime library and object files considerations

Both in version 3.x and version 4.x, two sets of runtime libraries are provided—CLIB and DLIB. In version 4.x, CLIB corresponds to the CLIB runtime library provided with version 3.x, and it can be used in the same way as before. DLIB, however, has been improved so that you can configure it to contain the features that are needed by your application.

To build code produced by version 4.x of the compiler, you must use the runtime environment components it provides. It is not possible to link object code produced using version 4.x with components provided with version 3.x.

For information about how to migrate from CLIB to DLIB, see *Migrating from CLIB to DLIB*, page 3. For more information about the two libraries, and the runtime environment they provide see the *AVR® IAR C/C++ Compiler Reference Guide*.

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 AVR 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 might use the `fprintf` function for terminal I/O (`stdout`), but might 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`. In other words, 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 defines the library functionality.

When you build an application using the IAR Embedded Workbench IDE, 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 you build an application from the command line, you must use the same library configuration file as when the library was built. For the prebuilt libraries (`r90`) there is a corresponding library configuration file (`h`), which has the same name as the library. The files are located in the `avr\lib` directory.

The command lines for specifying the library configuration file and library object file could look like this:

```
iccavr -dlib_config <install_dir>\avr\lib\dlib\dlavr-3s-ec-f.h
xlink dlavr-3s-ec-f.r90
```

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

To take advantage of the new features, it is recommended that you read about the runtime environment in the *AVR® 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.

The new linker option **Entry label** (`-s`) specifies 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. As before, any program modules containing a root segment part will also be loaded.

In version 4.x, the default program entry label in `cstartup.s90` 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.s90`.

If you build your application in the IAR Embedded Workbench IDE, you might simply add a 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 you link a C/C++ application. If you link without the option, the resulting output executable file will be empty, because no modules were referred to.

MIGRATING FROM CLIB TO DLIB

There are some considerations to have in mind if you want to migrate from the CLIB, the legacy C library, to the modern DLIB C/EC++ 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 IAR Embedded Workbench IDE to use the DLIB library.

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

Migrating from EWA90 to EWAVR

This chapter contains information that is useful when migrating from the A90 IAR Compiler (the EWA90 Compiler) to the AVR IAR C/C++ Compiler (the EWAVR Compiler). It briefly describes both differences and similarities between the products.

C source code that was originally written for the A90 IAR Compiler can be used also with the AVR IAR C/C++ Compiler, although some modifications may be required.

Note that if you are migrating from AVR IAR Embedded Workbench version 1.x (EWA90), you must also read the chapter *Migrating from version 3.x to version 4.x*.

Introduction

The main difference between the EWA90 Compiler and the EWAVR Compiler is that the latter is based on new compiler technology, which makes it possible to enhance your application code in a way that previously was not possible.

The most obvious difference is that with the new compiler technology, support for Embedded C++ has become available. Other main advantages include a new global optimizer, which improves the efficiency of the generated code. The consistency of the compiler is also improved due to the new technology.

Moreover, the new compiler technology allows you to write source code that is easily portable since it adheres more strictly to the ISO/ANSI standard; for example, it is possible to use pragma directives instead of extended keywords for defining special function registers (SFRs).

Also the checking of data types adheres more strictly to the ISO/ANSI standard in the EWAVR Compiler than in products using a previous compiler technology. You have the opportunity to identify and correct problems in the code, which improves the quality of the object code. Therefore, it is important to be aware of the fact that code written for the EWA90 Compiler may generate warnings or errors in the EWAVR Compiler.

The migration process

In short, to migrate from EWA90 to EWAVR, you must consider the following:

- The project file and project setup
- The C source code

To migrate your old project follow the described migration process. Note that not all steps 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

The workspace and projects you have created with EWA90 are not compatibe with EWAVR. If you are using the IAR Embedded Workbench IDE, follow these steps to convert your project file manually:

1. Start your new version of the AVR IAR Embedded Workbench IDE and create a new workspace by choosing **File>New** and then **Workspace**.
2. Choose **Project>Create New Project** to create a new project. To add your source code files, choose **Project>Add files**. For detailed information about how to create workspace and projects, see the tutorials available in the *AVR IAR Embedded Workbench User Guide*.
3. Because the available compiler options differ between EWA90 and EWAVR, you should verify your option settings.

   To generate a text file with the command line equivalents of the project options in your old project, see *Migrating project options*, page 6.

   Also, set any new options.
4. If you have your own customized 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. Replace EWA90 Compiler segments with EWAVR Compiler segments in the linker command file.

MIGRATING PROJECT OPTIONS

Because the available compiler options differ between EWA90 and EWAVR, you should verify your option settings after you have created your new project.

If you are using the command line interface, you can simply compare your makefile with the option tables in this chapter, see *Compiler options*, page 13, and modify the makefile accordingly.

If you are using the IAR Embedded Workbench IDE, follow these steps:

1. Open the old project in the old IAR Embedded Workbench IDE.
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 chapter, see *Compiler options*, page 13, 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 *AVR® IAR Embedded Workbench® IDE User Guide*.

C source code

Use the file `comp_a90.h` during the migration process. This file, which is provided with the product, contains translation macros that facilitate the migration.

In short, the process of migrating from the EWA90 Compiler to the EWAVR Compiler involves the following steps:

1. Replace EWA90 Compiler extended keywords in the source code with EWAVR Compiler keywords; see *Extended keywords*, page 8.
2. Replace EWA90 Compiler pragma directives with EWAVR Compiler directives. Notice that the behavior differs between the two products; see *Pragma directives*, page 10 for detailed information.
3. Replace EWA90 Compiler intrinsic functions with EWAVR Compiler intrinsic functions; see *Intrinsic functions*, page 12.
4. Transfer all relevant changes made to the `cstartup.s90` used in your EWA90 Compiler project and create a new `cstartup.s90` based on the file supplied with the EWAVR Compiler. It is not possible to use a `cstartup.s90` written for the EWA90 Compiler together with the EWAVR Compiler.

   The following sections describe the differences between the EWA90 Compiler and the EWAVR Compiler in detail.
5. To read about changed behavior of `sizeof` in preprocessor directives, see *Sizeof in preprocessor directives*, page 18.

Extended keywords

The set of language extensions has changed in the EWAVR Compiler. Some extensions have been added, some extensions have been removed, and for some of them the syntax has changed. There is also a rare case where an extension has a different interpretation if `typedefs` are used. This is described in the following section.

In the EWAVR Compiler, all extended keywords except `asm` start with two underscores, for example `__near`.

STORAGE MODIFIERS

Both the EWA90 Compiler and the EWAVR Compiler allow keywords that specify memory location. Each of these keywords 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 keywords are used directly in the source code, they behave in a similar way in the EWA90 Compiler and the EWAVR Compiler. The usage of type definitions and extended keywords is, however, more strict in the EWAVR Compiler than in the EWA90 Compiler.

Products based on the EWA90 Compiler behave unexpectedly in some cases:

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

The first variable declaration works as expected, that is `a` and `b` are located in near memory. The declaration of `c` is however illegal, except when near is the default memory, in which case there is no need for an extended keyword in the `typedef`.

In the last declaration, the `near` keyword of the `typedef` affects the location of the pointer variable `p`, not the pointer type. The pointer type is the default, which is given by the memory model.

The corresponding example for the EWAVR Compiler is:

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

The declarations of `c` and `p` differ. The `__huge` 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`. The location of the variable `p` is however not affected.

__NO_INIT

In the EWA90 Compiler, the keyword `no_init` is used for specifying that an object is not initialized. In the EWAVR Compiler `__no_init` can be used together with a keyword specifying any memory location, for example:

```
__near __no_init char buffer [1000];
```

__INTERRUPT

In the EWA90 Compiler, a vector can be attached to an interrupt function with the `#pragma function` directive or directly in the source code, for example:

```
interrupt [8] void f(void);
```

In the EWAVR Compiler a vector can be attached to an interrupt function with the `#pragma vector` directive, for example:

```
#pragma vector=8
__interrupt void f(void);
```

__MONITOR

In the EWA90 Compiler, the keyword `monitor` specifies not only the type attribute setting but also the memory location. In the EWAVR Compiler `__monitor` is an object attribute only.

SFR

In the EWA90 Compiler, the keywords `sfrb` and `sfrw` 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 PORT=0x10;
```

In the EWAVR Compiler the keywords `sfrb` and `sfrw` are not available. Instead you can:

- Place any object into any memory, by using a memory attribute; for example:
  `__io int b;`
- Locate any object at an absolute address by using the `#pragma location` directive or by using the locator operator `@`; for example:
  `long PORT @ 100;`

- Use the `volatile` attribute on any type, for example:
  `volatile __io char PORT@100;`

See the *AVR® IAR C/C++ Compiler Reference Guide* for complete information about the extended keywords available in the EWAVR Compiler.

Pragma directives

The EWA90 Compiler and the EWAVR Compiler have different sets of pragma directives for specifying attributes, and they also behave differently:

- In the EWA90 Compiler, the pragma directives change the default attribute to use for declared objects; they do not have an effect on pointer types. These directives are `#pragma memory` that specifies the default location of data objects, and `#pragma function` that specifies the default location of functions.
- In the EWAVR Compiler the pragma directives `type_attribute` and `object_attribute` change the next declared object or `typedef`.

The following EWA90 Compiler pragma directives have been removed in the EWAVR Compiler:

```
codeseg
function
warnings
```

They are recognized and will give a diagnostic message but will not work in the EWAVR Compiler.

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

The following table shows the mapping of pragma directives:

EWA90 Compiler pragma directive	EWAVR Compiler pragma directive
`#pragma function=interrupt`	`#pragma type_attribute=__interrupt` `#pragma vector=`*`long_word`* `offset`
`#pragma function=C_task`	`#pragma object_attribute=__c_task`
`#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`

Table 1: Mapping of pragma directives

EWA90 Compiler pragma directive	EWAVR Compiler pragma directive
`#pragma memory=far`	`#pragma type_attribute=__far`, `#pragma location`
`#pragma memory=flash`	`#pragma type_attribute=__flash`, `#pragma location`
`#pragma memory=huge`	`#pragma type_attribute=__huge`
`#pragma memory=near`	`#pragma type_attribute=__near`
`#pragma memory=no_init`	`#pragma object_attribute=__no_init`
`#pragma memory=tiny`	`#pragma type_attribute=__tiny`

Table 1: Mapping of pragma directives

It is important to notice that the EWAVR Compiler 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;
```

The EWAVR Compiler directives `#pragma constseg` and `#pragma dataseg` are active until they are explicitly turned off with the directive `#pragma constseg=default` and `#pragma dataseg=default`, respectively. For example:

```
#pragma constseg=myseg
__no_init f;
#pragma constseg=default
```

The EWAVR Compiler directive `#pragma memory=__xxxx` is active until it is explicitly turned off with the `#pragma memory=default` directive, for example:

```
#pragma memory=__near
int x,y,z;
#pragma memory=default
int myfunc()
```

The following pragma directives are identical in the EWA90 Compiler and the EWAVR Compiler:

```
#pragma language=extended
#pragma language=default
```

The following pragma directives have been *added* in the EWAVR Compiler:

```
#pragma constseg
#pragma dataseg
```

```
#pragma diag_default
#pragma diag_error
#pragma diag_remark
#pragma diag_suppress
#pragma diag_warning
#pragma location
#pragma object_attribute
#pragma optimize
#pragma pack
#pragma type_attribute
#pragma vector
```

Specific segment placement

In the EWA90 Compiler, the `#pragma memory` directive supports a syntax enabling 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` they must not be declared `const`.

In the EWAVR Compiler, the directive `#pragma location` and the `@` operator are available for this purpose.

See the *AVR® IAR C/C++ Compiler Reference Guide* for detailed information about the pragma directives available in the EWAVR Compiler.

Predefined symbols

In both the EWA90 Compiler and the EWAVR Compiler, all predefined symbols start with two underscores, for example `__IAR_SYSTEMS_ICC__`. Note however that in the EWAVR Compiler all but two predefined symbols also end with two underscores.

See the *AVR® IAR C/C++ Compiler Reference Guide* for complete information about the predefined symbols available in the EWAVR Compiler.

Intrinsic functions

In the EWAVR Compiler, the intrinsic functions start with two underscores, for example `__enable_interrupt`.

The EWA90 Compiler intrinsic functions `_args$` and `_argt$` are not available in the EWAVR Compiler. Other EWA90 Compiler intrinsic functions have equivalents with other names in the EWAVR Compiler.

See the *AVR® IAR C/C++ Compiler Reference Guide* for complete information about the intrinsic functions available in the EWAVR Compiler.

Compiler options

COMMAND LINE SYNTAX

The command line options in the EWAVR Compiler 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 `--cpu=2343`. This is the preferred style in the EWAVR Compiler.
- Short option names consisting of a single letter prefixed with a single dash and sometimes followed by a modifier, for example `-r` or `-mt`. This style is available in the EWAVR Compiler mainly for backward compatibility.

Some options appear in one style only, other options appear in both styles.

Removed EWA90 options

The following table shows the EWA90 Compiler command line options that have been removed in the EWAVR Compiler:

EWA90 Compiler option	Description
`-C`	Nested comments
`-E`	Constants and string literals in flash
`-F`	Form-feed after each function
`-G`	Open standard input as source. Replaced by `-` (dash) as source file in the EWAVR Compiler.
`-g`	Global strict type check. In the EWAVR Compiler, global strict type checking is always enabled.
`-gO`	No type information in object code
`-K`	`'//'` comments. In the EWAVR Compiler, `'//'` comments are allowed unless the option `--strict_ansi` is used.
`-O`*`prefix`*	Set object filename prefix. In the EWAVR Compiler, use `-o` *`filename`* instead.

Table 2: EWA90 Compiler options removed from EWAVR Compiler

EWA90 Compiler option	Description
-P	Generate PROMable code. This functionality is always enabled in the EWAVR Compiler.
-p*nn*	Lines/page
-T	Active lines only
-t	Tab spacing
-U*symb*	Undefine symbol
-X	Explain C declarations
-x[DFT2]	Cross-reference
-Y	Writable strings

Table 2: EWA90 Compiler options removed from EWAVR Compiler (Continued)

Note: The EWA90 Compiler command line option -f was not supported by the first versions of the EWAVR Compiler, but has been available again since version 2.26A.

Identical options

The following table shows the command line options that are *identical* in the EWA90 Compiler and the EWAVR Compiler:

Option	Comment
-D*symb*=*value*	Define symbols
-e	Language extensions
-I	Include paths. (Syntax is more free in the EWAVR Compiler.)
-o *filename*	Set object filename
-s[0-9]	Optimize for speed
-z[0-9]	Optimize for size

Table 3: Identical options in the EWA90 Compiler and the EWAVR Compiler

Renamed or modified EWA90 Compiler options

The following EWA90 Compiler command line options have been *renamed* and/or *modified*:

EWA90 Compiler option	EWAVR Compiler option	Description
`-A` `-a` *filename*	`-la .` `-la` *filename*	Assembly output. See *Filenames*, page 16.
`-b`	`--library_module`	Make object a library module
`-c`	`--char_is_signed`	`'char'` is `'signed char'`
`-gA`	`--strict_ansi`	Flag old-style functions
`-H`*name*	`--module_name=`*name*	Set object module name
`-L[`*prefix*`],` `-l[c|C|a|A][N]` *filename*	`-l[c|C|a|A][N]` *filename*	List file. The modifiers specify the type of list file to create.
`-N`*prefix*`, -n`	`--preprocess=[c][n][l]` *filename*	Preprocessor output
`-q`	`-lA, -lC`	Insert mnemonics. List file syntax has changed.
`-R` *name*	`--segment`	Set code segment name.
`-r[012][i][n]`	`-r` `--debug`	Generate debug information. The modifiers have been removed.
`-S`	`--silent`	Set silent operation
`-v[0|1|2|3]`	`--cpu=`*xxxx* `-v[0|1|2|3|4|5|6]`	Processor configuration.
`-w`	`--no_warnings`	Disable warnings

Table 4: Renamed or modified EWA90 Compiler command line options

Note: A number of new command line options have been added in the EWAVR Compiler. For a complete list of the available command line options, see the *AVR® IAR C/C++ Compiler Reference Guide*.

FILENAMES

In the EWA90 Compiler, 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` (Assembly output to named file).
- With a prefix string added to the default name, using a command line option such as `-A[prefix]` (Assembly output to prefixed filename).

In the EWAVR Compiler, 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:

EWA90 Compiler command	EWAVR Compiler command	Result
`-l myfile`	`-l myfile`	`myfile.lst`
`-Lmyfile`	`-l myfile`	`myfile.lst`
`-L`	`-l .`	`test.lst`
`-Lmydir/`	`-l mydir`	`-l mydir/mydir/test.lst`

Table 5: Filenames in the EWA90 Compiler and the EWAVR Compiler

LIST FILES

In the EWA90 Compiler, only one C list file and one assembler list file can be produced; in the EWAVR Compiler there is no upper limit on the number of list files that can be generated. The EWAVR Compiler command line option `-l[c|C|a|A][N] filename` is used for specifying the behavior of each list file.

OBJECT FILE FORMAT

In some products using the previous generation of compiler technology, 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, and when the command line option `-re` is used, the actual source code is embedded in the object format.

In the EWAVR Compiler, when the command line option `-r` or `--debug` is used, source file references are always generated, i.e. embedding of the source code is not supported.

NESTED COMMENTS

In the EWA90 Compiler, nested comments were allowed if the option `-C` was used. In the EWAVR Compiler, nested comments are never allowed. For example, if a comment were 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.

PREPROCESSOR FILE

In the EWA90 Compiler, a preprocessor file can be generated as a side effect of compiling a source file.

In the EWAVR Compiler, a preprocessor file is either generated as a side effect, or as the whole purpose when parsing of the source code is not required. You may also choose to include or exclude comments and/or `#line` directives.

CROSS-REFERENCE INFORMATION

In the EWA90 Compiler, cross-reference information can be generated. This possibility is not available in the EWAVR Compiler.

SIZEOF IN PREPROCESSOR DIRECTIVES

In the EWA90 Compiler, `sizeof` could be used in `#if` directives, for example:

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

In the EWAVR Compiler, `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:

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

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

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

1 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));
}
```

2 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);
```

Note: The file `limits.h` contains macro definitions that can be used instead of `#if sizeof`.