Beyond Hello World - Advanced Arm Compiler 5 Features

Version 1.0

Non-Confidential

Issue 01

102748_0100_01_en

Beyond Hello World - Advanced Arm Compiler 5 Features

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	12 November 2021	Non-Confidential	Initial release

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction

The Building hello world using Arm Compiler tutorial shows you how to build a simple C program with the Arm Compiler 5 toolchain.

This tutorial moves beyond the basics to explore some of the more advanced features of the Arm Compiler 5 toolchain.

This tutorial assumes you have installed and licensed Arm DS-5 Development Studio. For more information, see Getting Started with Arm DS-5 Development Studio.

2. Compiling mixed C and assembly source files

The Arm assembler, `armasm` reads assembly language source code and outputs object code.

The Arm compiler, `armcc` compiles C and C++ source to object code.

The Arm linker, `armlink` combines the contents of one or more object files with any required libraries to produce an executable program.

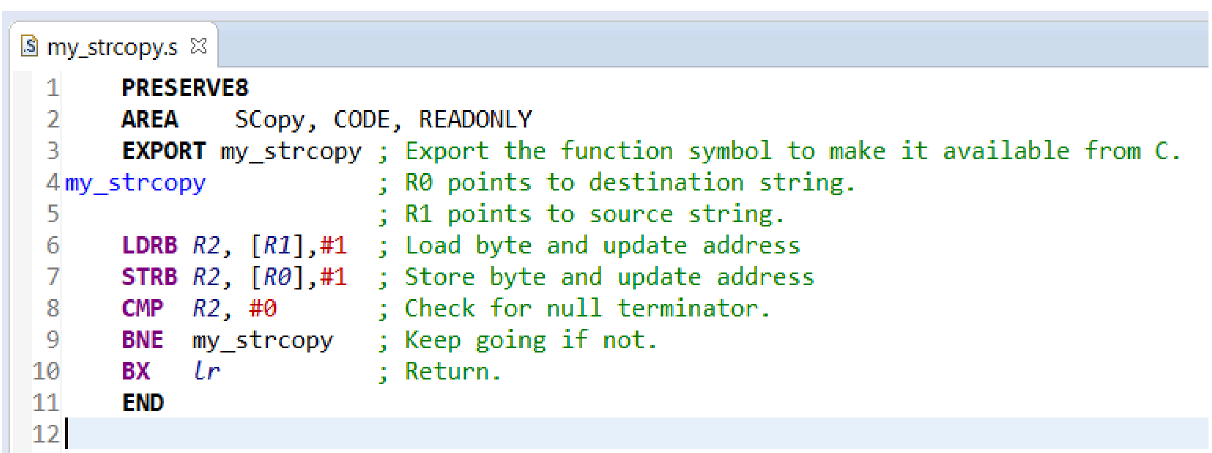
The following example shows how to use `armasm`, `armcc`, and `armlink` from DS-5 to build a project containing both C and assembly source files.

1. Create a new C project and add a new source file `my_strcopy.s` containing the following assembly code:

```
    PRESERVE8
    AREA    SCopy, CODE, READONLY
    EXPORT my_strcopy ; Export symbol
my_strcopy            ; R0 -> dest string
                      ; R1 -> source string
    LDRB R2, [R1],#1  ; Load byte + update addr
    STRB R2, [R0],#1  ; Store byte + update addr
    CMP  R2, #0       ; Check for null
    BNE  my_strcopy   ; Keep going if not
    BX   lr           ; Return
    END
```

The function `my_strcopy()` is exported so that it is available to be used from C.

Figure 2-1: An image of an assembly source.

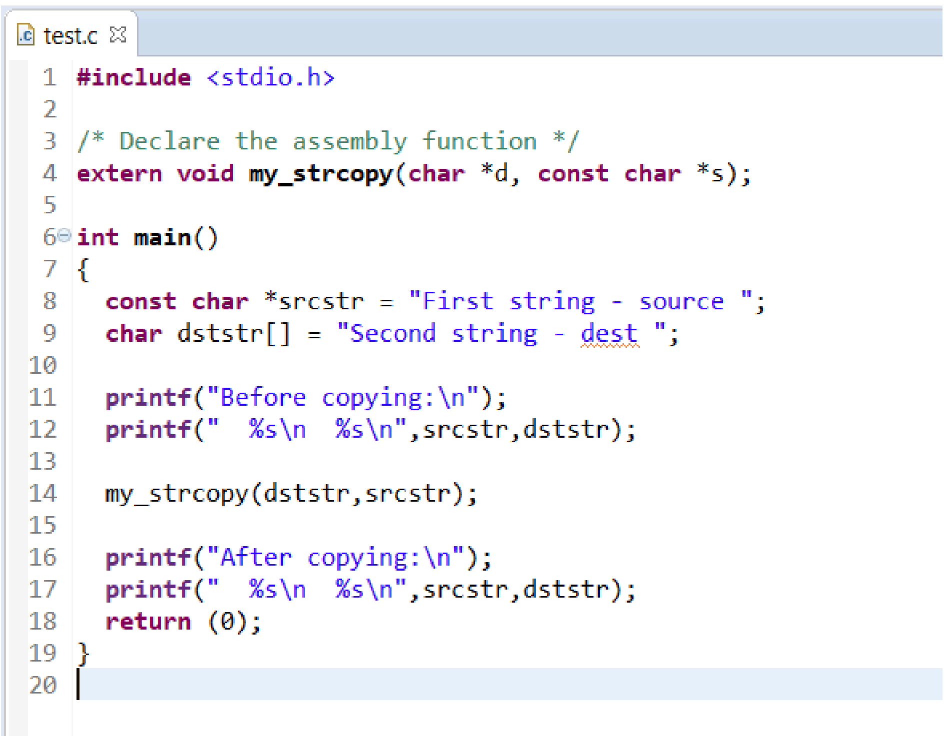


2. Add a new source file to the project with the name `test.c` containing the following C code:

```
#include <stdio.h>

/* Declare the assembly function */
extern void my_strcopy(char *d, const char *s);
```

```
int main()
{
  const char *srcstr = "First string - source ";
  char dststr[] = "Second string - dest ";

  printf("Before copying:\n");
  printf("  %s\n  %s\n",srcstr,dststr);

  my_strcopy(dststr,srcstr);

  printf("After copying:\n");
  printf("  %s\n  %s\n",srcstr,dststr);
  return (0);
}
```

Figure 2-2: An image of a c source.

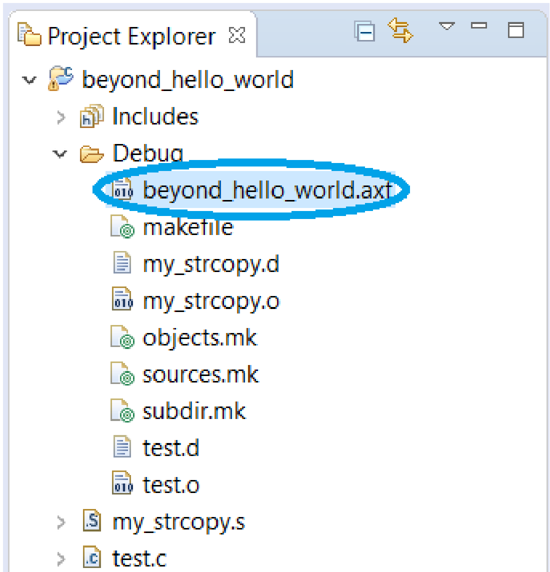


3. Build the project. The Arm Compiler toolchain does the following:
   a. Assembles my_strcopy.s with armasm to produce the object file my_strcopy.o.
   b. Compiles test.c with armcc to produce the object file test.o.
   c. Links the object files with armlink to produce an executable image.

   When you run the executable image, it produces the following output:

```
Before copying:
  First string - source
  Second string - dest
After copying:
  First string - source
  First string - source
```

Figure 2-3: A compiled image.



3. Sharing header files between C and assembly code

The usual way to define constants in C code is to use `#define`-s, or in assembly code to use `EQU` directives. If your project contains a mixture of C and assembly code, there might be some constant definitions that are common to both. If so, to avoid maintaining two separate lists, you can create one list of common definitions and include them in both your C and assembly code.

To do this, you can use C-style `#include` and `#define` directives directly in your assembly source code. You can pass this source code through the `armcc` C preprocessor. This outputs a preprocessed version of your assembly code which `armasm` can then assemble.

The following example shows how to do this.

1. Add a header file called `my_strcopy.h` to the project, containing the following line:

```
#define ONE_CONSTANT 1
```

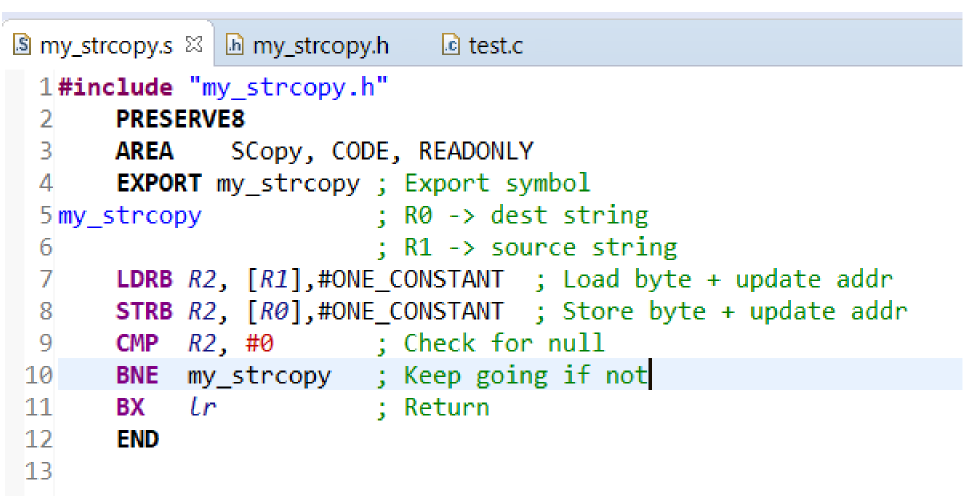
2. Add this line to the top of `my_strcopy.s`, created in the previous example:

```
#include "my_strcopy.h"
```

3. In `my_strcopy.s`, replace the occurrences of `#1` with `#ONE_CONSTANT`, for example:

```
LDRB R2, [R1], #ONE_CONSTANT
```

Figure 3-1: A image of an assembly source.



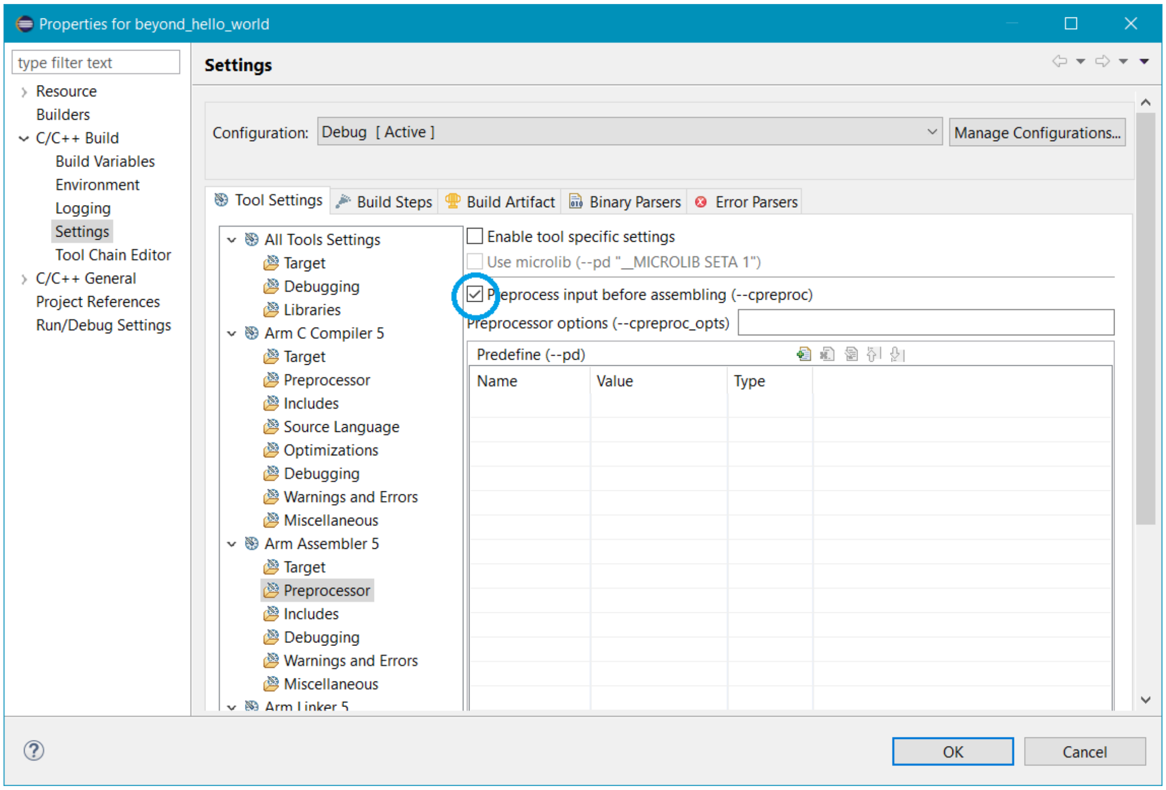
4. Pass `my_strcopy.s` through the C preprocessor. If you tried to build the project without first doing this, `armasm` would report a syntax error for the `#include` statement you added to `my_strcopy.s`.

    a. Open the Project Settings dialog.

b. Under C/C++ build, select Settings.

c. In the Tool Settings tab, under Arm Assembler 5, select Preprocessor.

d. Tick the box marked `Preprocess input before assembling (--cpreproc)`.



`armasm` automatically passes some command-line options to the C preprocessor. If you need to pass other simple command-line options to the C preprocessor, for example -D or -I, specify them in the field marked Preprocessor options (–cpreproc_opts).

4. Improving optimization with linker feedback

Linker feedback lets the compiler and linker collaborate to improve the removal of unused code.

The linker can produce a text file containing a list of unused functions and functions that have been inlined. This information can be fed back to the compiler, which rebuilds the objects, placing these functions in their own sections. These sections can then be removed by the linker during usual unused section elimination.

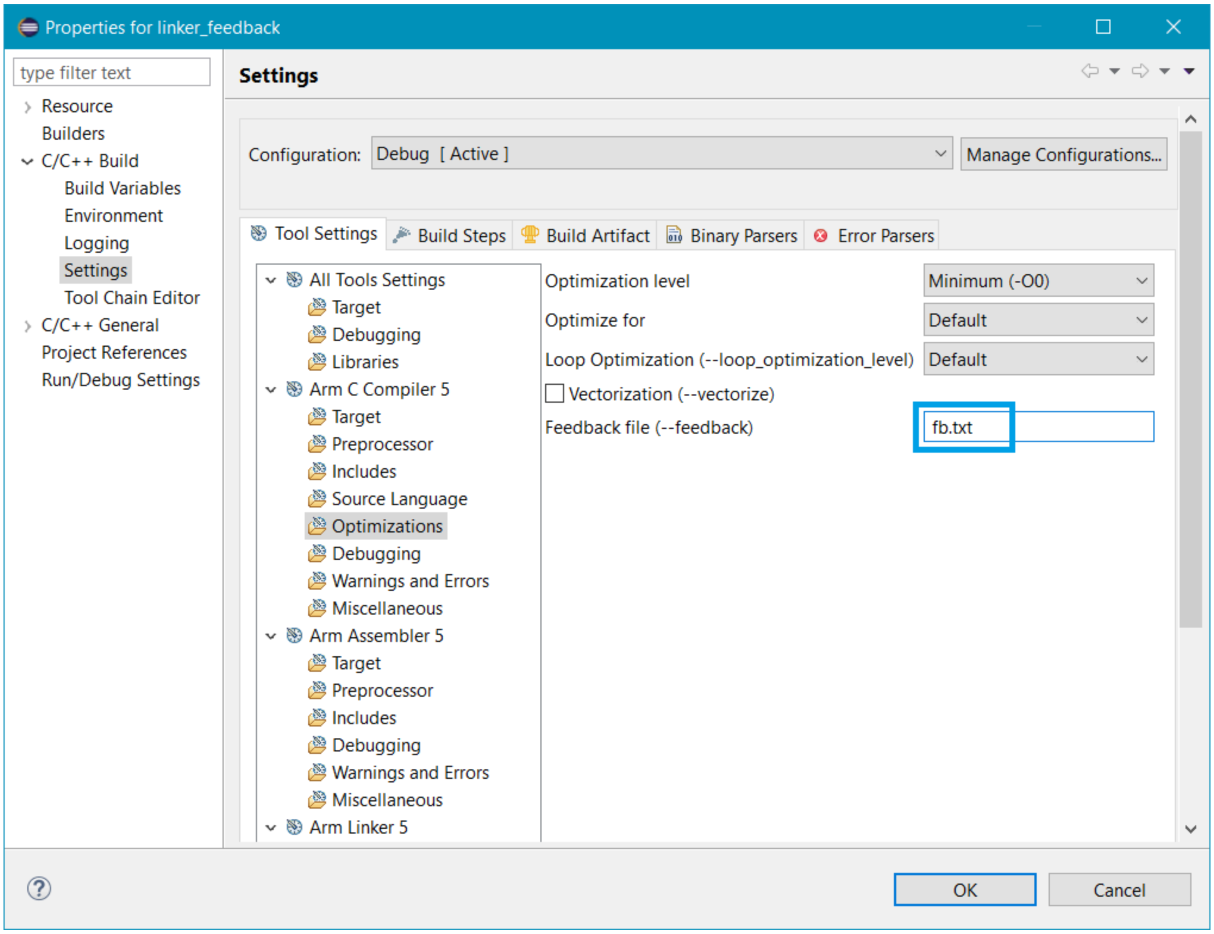
The following example shows how linker feedback works.

1. Create a new C project and add a new source file `fb.c` containing the following code:

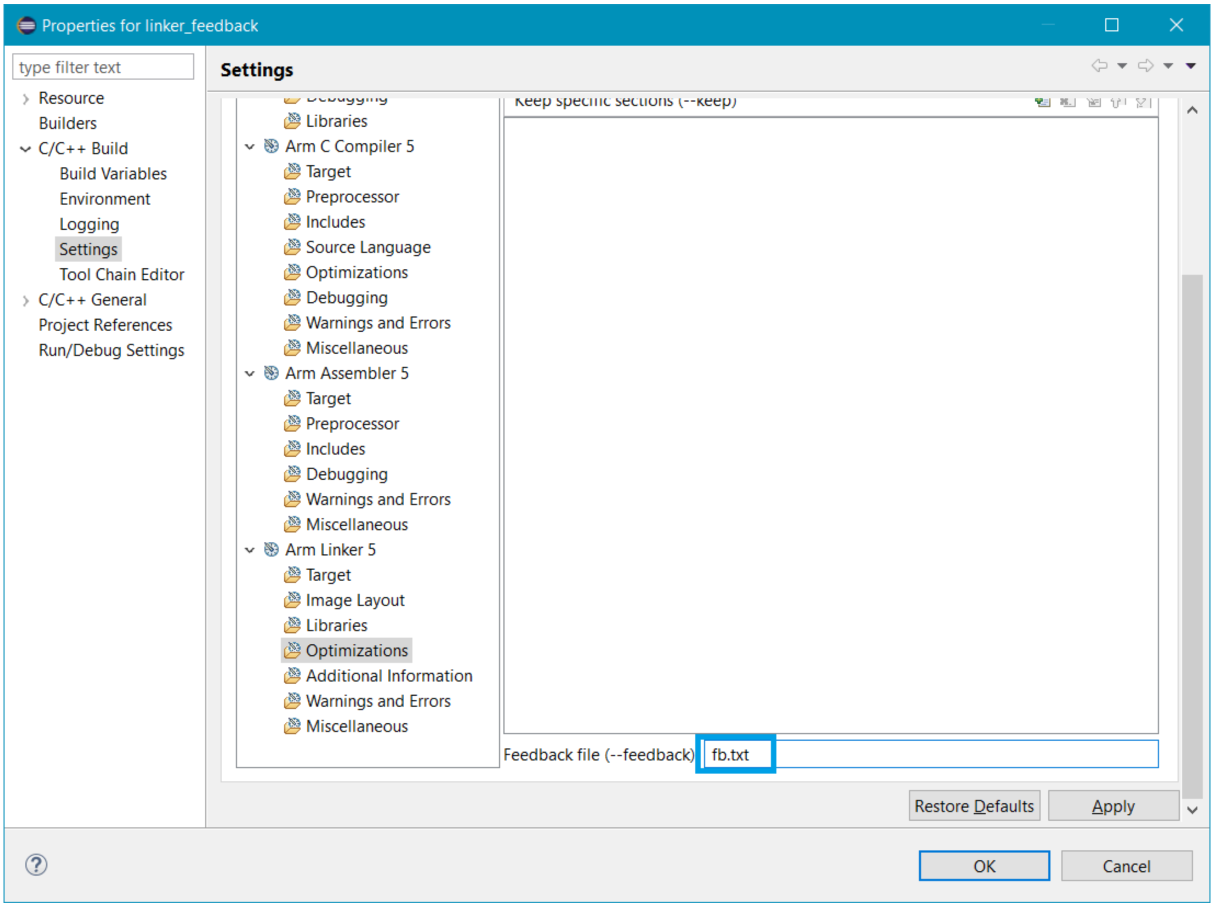
```
#include <stdio.h>
    void legacy()
     {
       printf("This is an unused function.\n");
     }

    int cubed(int i)
     {
       return i*i*i;
     }

    int main(void)
     {
      int n = 3;
      printf("%d cubed = %d\n",n,cubed(n));
     }
```

2. Open the Properties dialog box for your project, and select C/C++ Build > Settings.

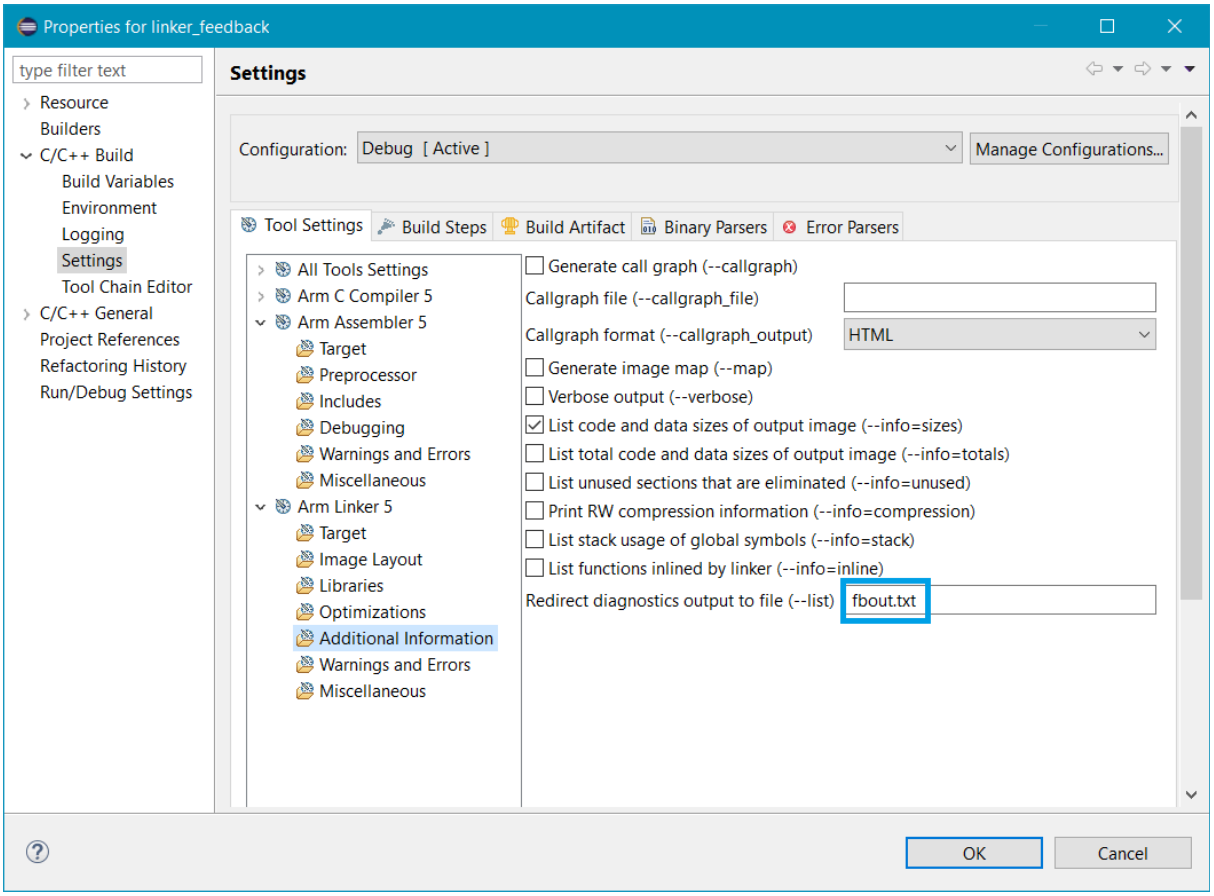
3. On the Settings tab, select Arm C Compiler 5 > Optimizations and specify “Feedback file (–feedback)” `fb.txt`. This tells the compiler to look for a feedback file.



4. On the Settings tab, select Arm Linker 5 > Optimizations and specify "Feedback file (–feedback)" `fb.txt`. This tells the linker to create a feedback file.

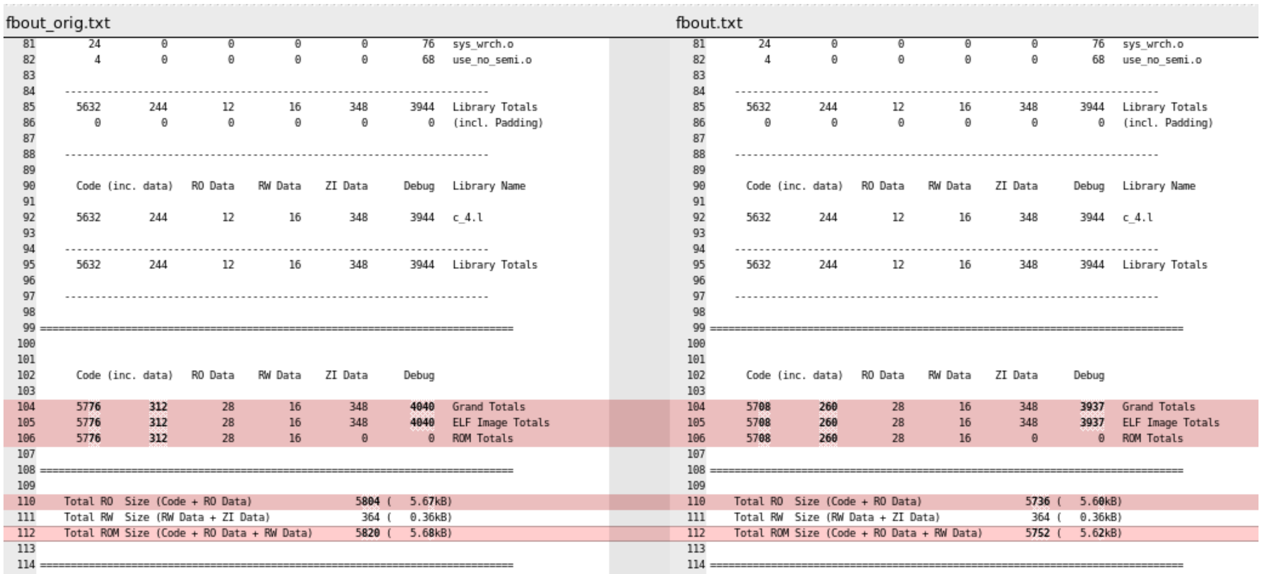


5. On the Tool Settings tab, select Arm Linker 5 > Additional Information and specify “Redirect diagnostics output to file (–list)” `fbout.txt`. This tells the linker to save diagnostics to a file.



6. Build the project. For this first compilation, the feedback file does not exist at compilation time. At link time, the linker identifies `legacy()` as the unused function, and creates a feedback file fb.txt containing this information.
7. Clean the project (Project > Clean...), to remove the object and image files.
8. Rename the diagnostics file `fbout.txt` to `fbout_orig.txt` to let you compare it to the next build.
9. Build the project again. This time, the feedback file does exist at compilation time. Because the feedback file informs the compiler that `legacy()` is an unused function, the compiler can now omit this function from the generated object code.

You can compare the two diagnostics files, `fbout.txt` and `fbout_orig.txt`, to see the sizes of the image components (for example, Code, RO Data, RW Data, and ZI Data). The Code component is smaller, because `armlink` has removed the `legacy()` function from the final image.



fbout_orig.txt

```
        24          0          0          0          0         76   sys_wrch.o
         4          0          0          0          0         68   use_no_semi.o

    ----------------------------------------------------------------------
      5632        244         12         16        348       3944   Library Totals
         0          0          0          0          0          0   (incl. Padding)

    ----------------------------------------------------------------------

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Library Name

      5632        244         12         16        348       3944   c_4.l

    ----------------------------------------------------------------------
      5632        244         12         16        348       3944   Library Totals

    ----------------------------------------------------------------------

==============================================================================


      Code (inc. data)   RO Data    RW Data    ZI Data      Debug

      5776        312         28         16        348       4040   Grand Totals
      5776        312         28         16        348       4040   ELF Image Totals
      5776        312         28         16          0          0   ROM Totals

==============================================================================

    Total RO  Size (Code + RO Data)                 5804 (   5.67kB)
    Total RW  Size (RW Data + ZI Data)               364 (   0.36kB)
    Total ROM Size (Code + RO Data + RW Data)       5820 (   5.68kB)

==============================================================================
```

fbout.txt

```
        24          0          0          0          0         76   sys_wrch.o
         4          0          0          0          0         68   use_no_semi.o

    ----------------------------------------------------------------------
      5632        244         12         16        348       3944   Library Totals
         0          0          0          0          0          0   (incl. Padding)

    ----------------------------------------------------------------------

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Library Name

      5632        244         12         16        348       3944   c_4.l

    ----------------------------------------------------------------------
      5632        244         12         16        348       3944   Library Totals

    ----------------------------------------------------------------------

==============================================================================


      Code (inc. data)   RO Data    RW Data    ZI Data      Debug

      5708        260         28         16        348       3937   Grand Totals
      5708        260         28         16        348       3937   ELF Image Totals
      5708        260         28         16          0          0   ROM Totals

==============================================================================

    Total RO  Size (Code + RO Data)                 5736 (   5.60kB)
    Total RW  Size (RW Data + ZI Data)               364 (   0.36kB)
    Total ROM Size (Code + RO Data + RW Data)       5752 (   5.62kB)

==============================================================================
```

5. Further reading

To read more about Arm Compiler 5, follow the links:

- –cpreproc
- –cpreproc_opts=option[,option,...]
- Using the C preprocessor

Arm Compiler `armcc` User Guide

- –feedback=filename
- About linker feedback
- Example of using linker feedback
- –feedback=filename