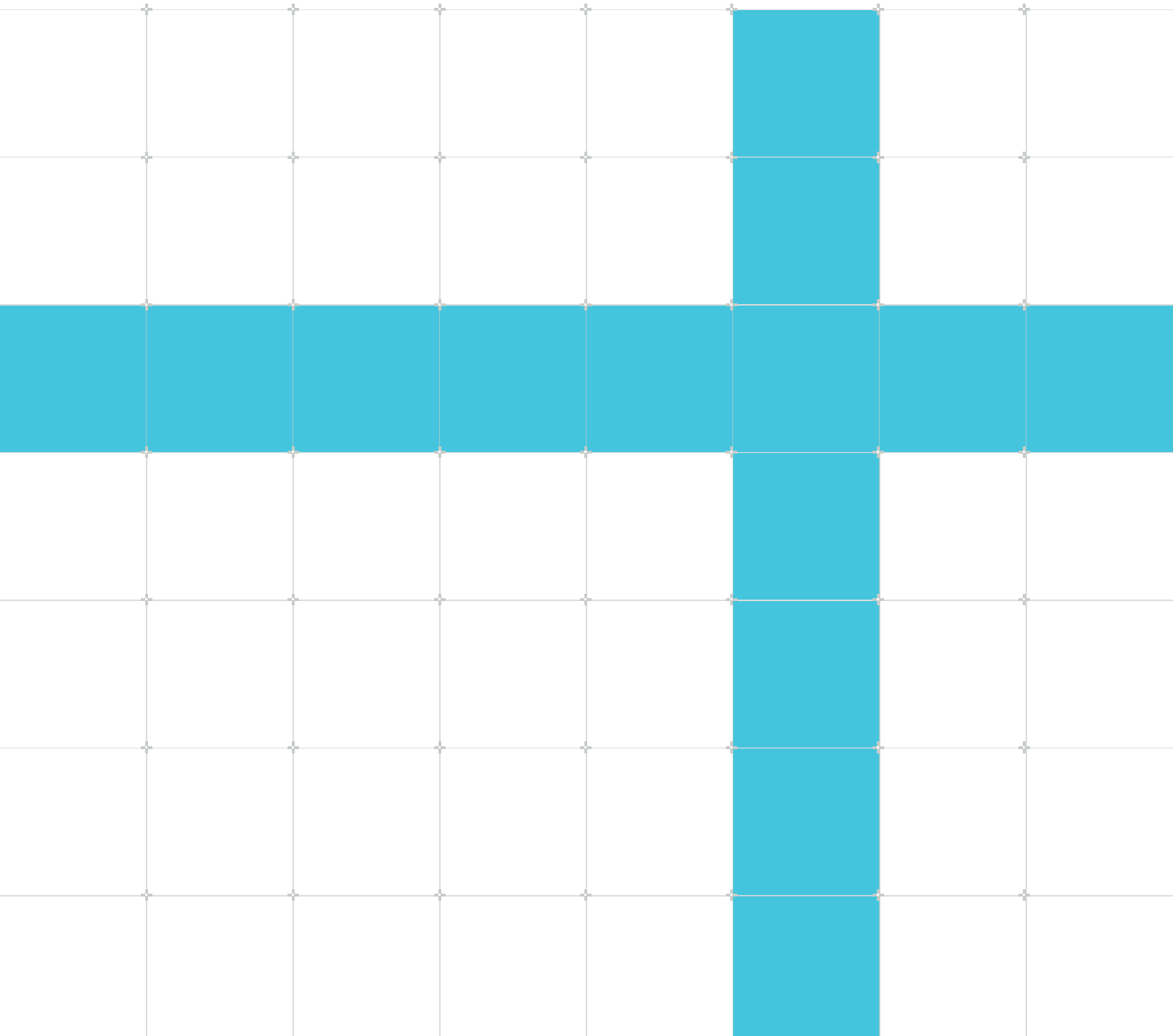
Beyond Hello World - advanced Arm Compiler 6 features

Version 1.0

Issue 02
102665_0100_02_en



Beyond Hello World - advanced Arm Compiler 6 features

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	22 September 2021	Non-Confidential	Initial release

Proprietary Notice

or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

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. Overview

The Building hello world using Arm Compiler 6 tutorial shows you how to build a simple C program with the Arm Compiler 6 toolchain.

This tutorial explores some of the more advanced features of the Arm Compiler 6 toolchain.

Before you begin

Install and license Arm DS Development Studio. For more information on installation and licensing, see Getting started with Arm Development Studio.

Arm Compiler 6 adopts the LLVM integrated assembler as default because it aligns more closely with GNU assembler syntax, improving portability between GNU and Arm Compiler toolchains. The LLVM integrated assembler is called by default by `armclang`. A side effect is that Arm Compiler does not compile C/C++ source files which contain legacy `armcc` inline or embedded assembler.

2. Compiling mixed C and assembly source files

The Arm assembler armclang is an integrated assembler that is based on LLVM using GNU syntax and reads assembly language source code and outputs object code. The Arm compiler armclang compiles C and C++ source code 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. Arm compiler, assembler, and linker are all part of the Arm Compiler 6 toolchain which is inbuilt with Arm DS.

The following example shows how to use armclang integrated assembler, armclang and armlink from Arm DS 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:

```
#include "my_strcpy.h"
    .section   StringCopy, "ax"
    .balign    8
    .global    mystrcopy
    .type      mystrcopy, "function"
    mystrcopy:
    ldrb       r2, [r1], #1                ; Load byte and update address
    strb       r2, [r0], #1                ; Store byte and update address
    cmp        r2, #0                      ; Check for null terminator
    bne        mystrcopy                   ; Keep going if not
    bx         lr                          ; Return
    .end
```

The function `my_strcopy()` is exported so that it is available to be used from C, see the following code screenshot:

Figure 2-1: 2-A-4-assembly source

```
 7
 8 #include "my_strcpy.h"
 9     .section   StringCopy, "ax"
10     .balign    8
11     .global    mystrcopy
12     .type      mystrcopy, "function"
13 mystrcopy:
14     ldrb       r2, [r1], #1            ; Load byte and update address
15     strb       r2, [r0], #1            ; Store byte and update address
16     cmp        r2, #0                  ; Check for null terminator
17     bne        mystrcopy               ; Keep going if not
18     bx         lr                      ; Return
19     .end
20
```

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

```
#include <stdio.h>
#include <stdlib.h>

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

int main()
{
  const char *srcstr = "First string - source ";
  char *dststr = "Second string - dest ";
  puts("Before copying:\n");
  printf("  %s\n  %s\n",srcstr,dststr);

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

Figure 2-2: 2-A-3-test code

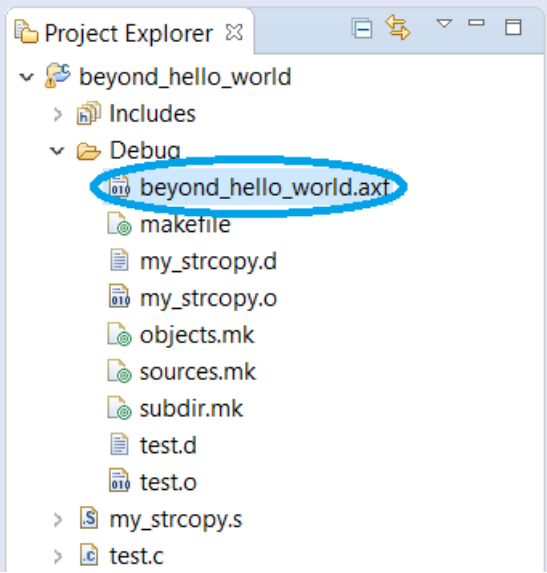
```
 9 #include <stdio.h>
10 #include <stdlib.h>
11
12 /* Declare the assembly function */
13 extern void mystrcopy(char *d, const char *s);
14
15 int main()
16 {
17   const char *srcstr = "First string - source ";
18   char *dststr = "Second string - dest ";
19   puts("Before copying:\n");
20   printf("  %s\n  %s\n",srcstr,dststr);
21
22   mystrcopy(dststr,srcstr);
23   puts("\nAfter copying:\n");
24   printf("  %s\n  %s\n",srcstr,dststr);
25   return (0);
26 }
27
```

3. Build the project. The Arm Compiler toolchain does the following:
   - Assembles `my_strcopy.s` with `armclang` `my_strcopy.o`.
   - Compiles `test.c` with `armclang` to produce the object file `test.o`.
   - 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: 2-A-3-compiled image



3. Sharing header files between C and assembly code

The usual way to define constants in C code is to use `#define`, or in assembly code to use `CMP` 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 make common definitions, you can use C-style `#include` and `#define` directives directly in your assembly source code. You can pass this source code through the `armclang` C preprocessor. It outputs a preprocessed version of your assembly code which `armclang` 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:

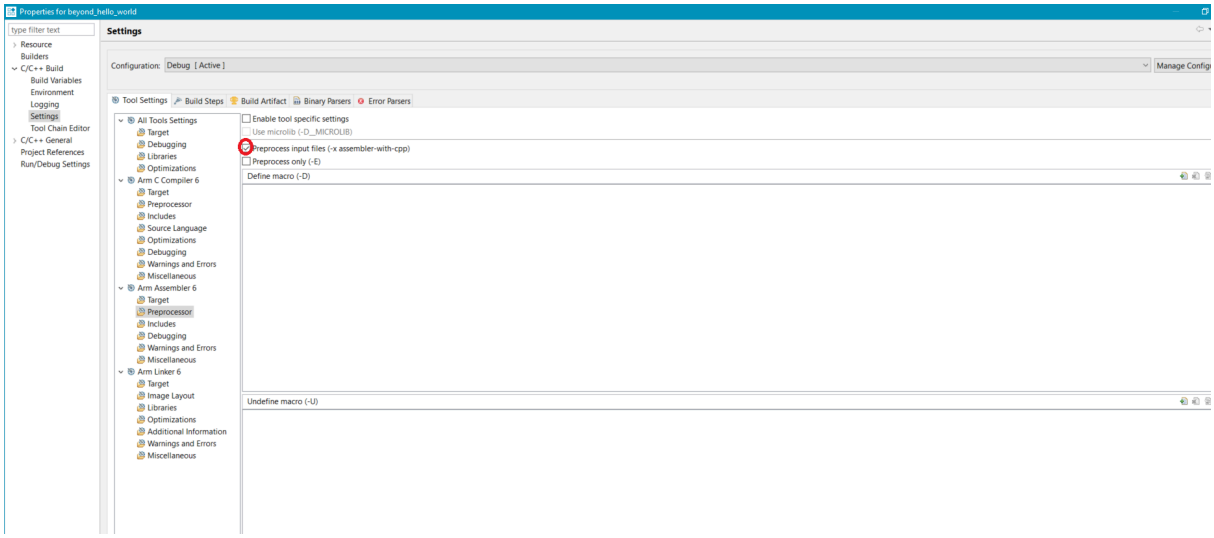
```
LDRB R2, [R1], #ONE_CONSTANT
```

Figure 3-1: 2-A-1-assembley source

```
 8 #include "my_strcpy.h"
 9     .section   StringCopy, "ax"
10     .balign    8
11     .global    mystrcopy
12     .type      mystrcopy, "function"
13 mystrcopy:
14     ldrb       r2, [r1], #ONE_CONSTANT ; Load byte and update address
15     strb       r2, [r0], #ONE_CONSTANT ; Store byte and update address
16     cmp        r2, #0                  ; Check for null terminator
17     bne        mystrcopy               ; Keep going if not
18     bx         lr                      ; Return
19     .end
20
```

4. Pass `my_strcopy.s` through the C preprocessor. If you tried to build the project without first doing this, Arm assembler 6 would report a syntax error for the `#include` statement you added to `my_strcopy.s`.
5. Open the Project Settings dialog. Then, under C/C++ build, select Settings. In the Tool Settings tab, under Arm Assembler 6, select Preprocessor, then tick the box marked Preprocess input before assembling (- x assembler-with-cpp), as shown in the following image:

Figure 3-2: 2-A-2-peprocessor setting



6. The -x assembler-with-cpp option tells armclang that the assembly source file requires preprocessing. Now if you try to build the project, it will be able to successfully build it.
7. If you need to pass other simple command-line options to the C preprocessor, for example -D,-U or -E, specify them in the field shown in Preprocessor window. Deatils of these options can be found in the Arm compiler 6 documentation.