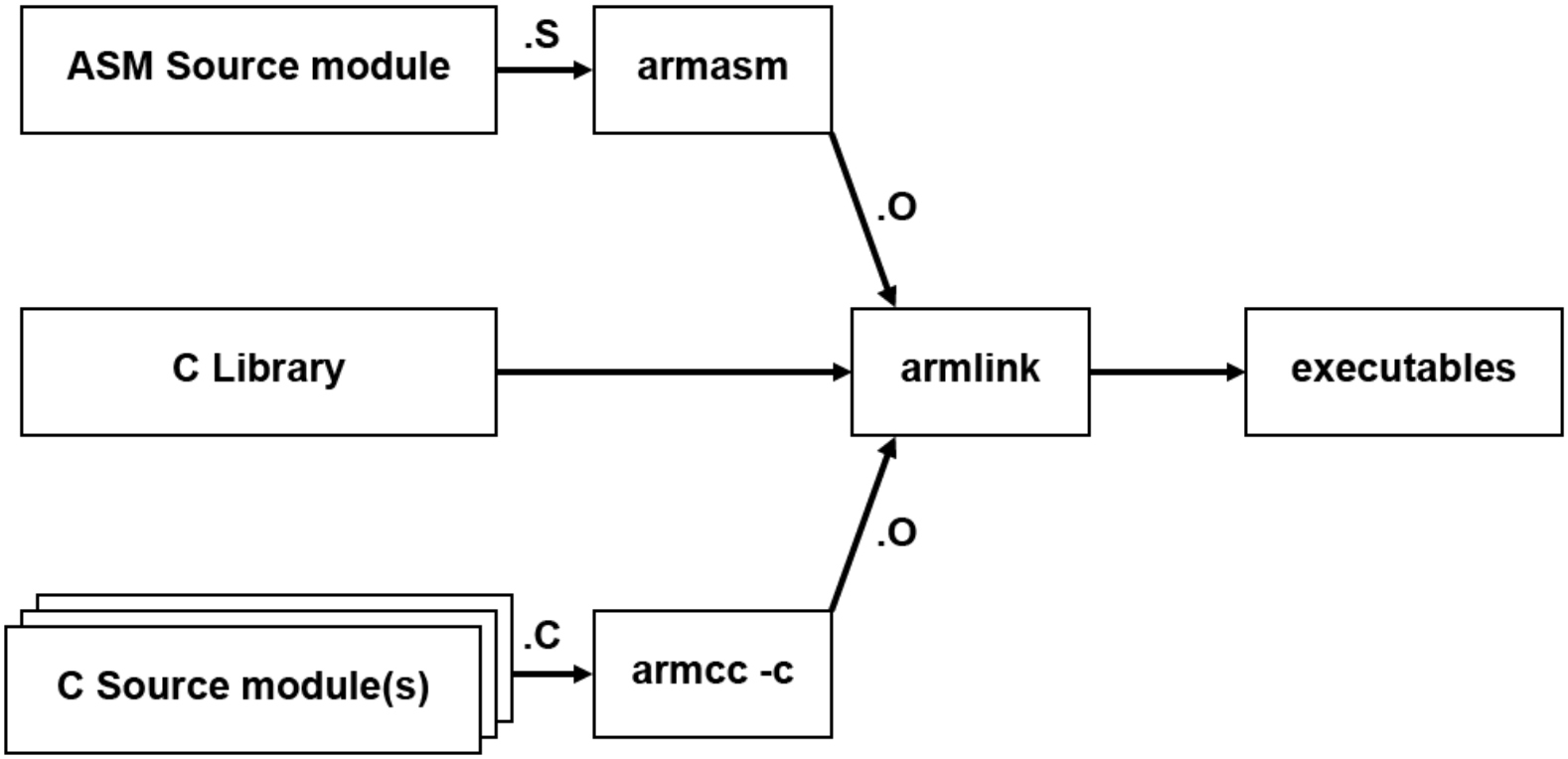
ESET 349 – Microcontroller Architecture

C- Assembly

Dr. Muhammad Faeyz Karim

Mixing C and Assembly



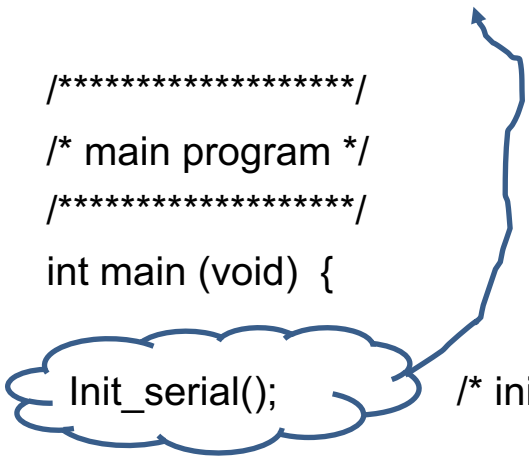
Why Mixed? Motivation?

- It is quite common especially in embedded applications
- Optimize certain critical codes for better performance
- 2 ways to add assembly to high level code
  - Inline assembler
  - Embedded assembler

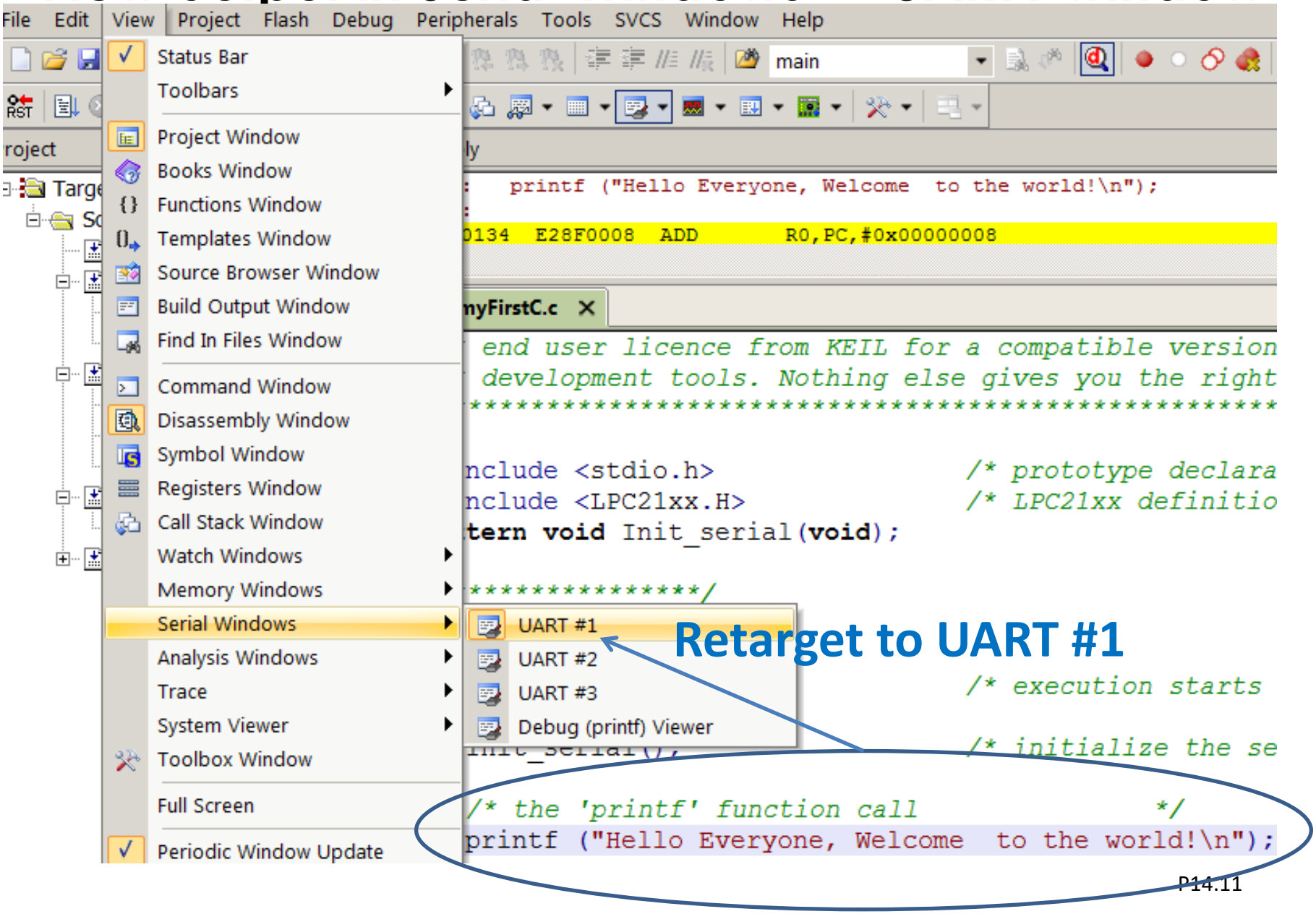
efficient

Example 1: Myfirstc.C

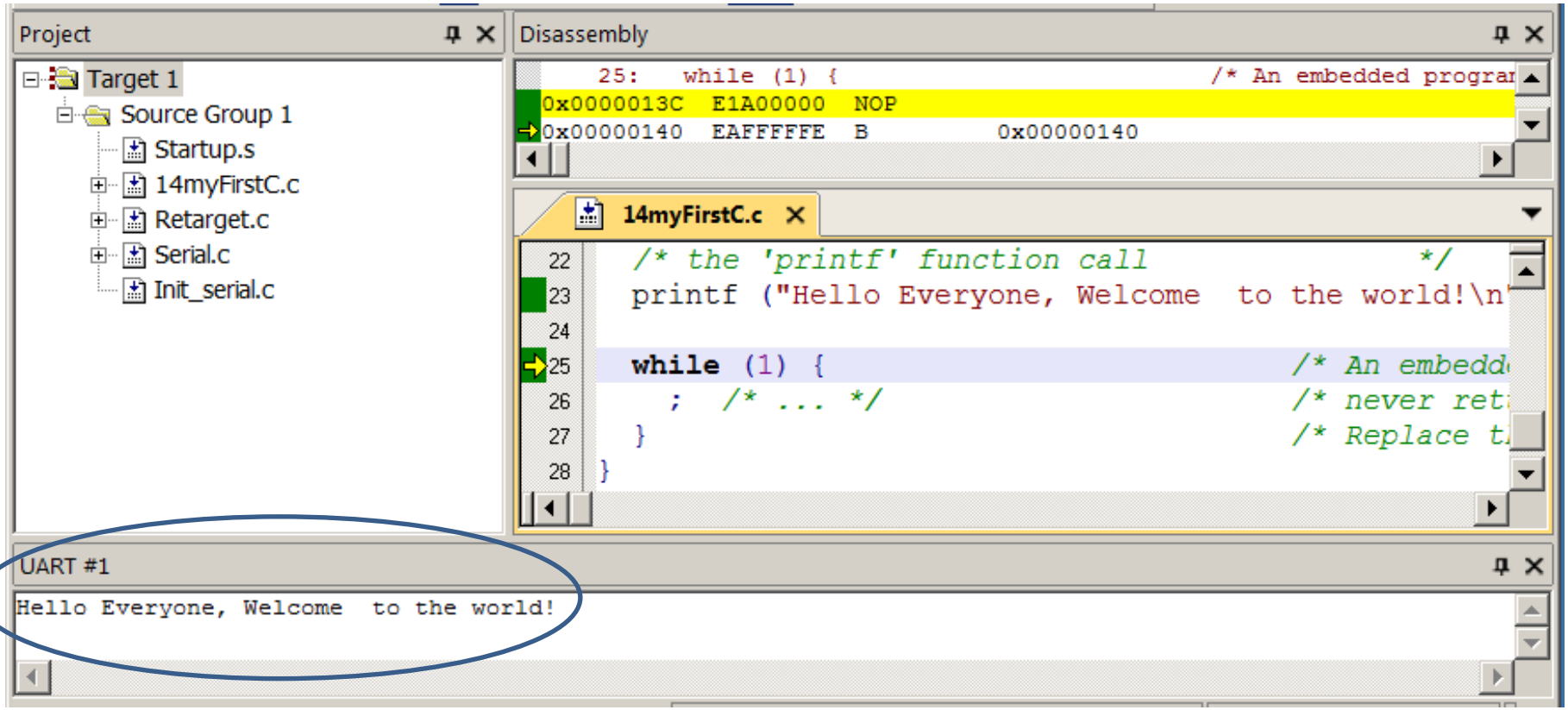
```
#include <stdio.h>                  /* prototype declarations for I/O functions */
#include <LPC21xx.H>                /* LPC21xx definitions                      */
extern void Init_serial(void);

/*******************/
/* main program */
/*******************/
int main (void)  {                  /* execution starts here                    */

  Init_serial();                    /* initialize the serial interface          */

   /* the 'printf' function call */
   printf ("Hello Everyone, Welcome  to the world!\n");

   while (1) {                      /* An embedded program does not stop and    */
     ;  /* ... */                   /* never returns. We use an endless loop.   */
   }                                /* Replace the dots (...) with your own code.*/
}
```



View output in Serial Windows → UART1 window



Demo 1: myFirstC.c



Retarget to UART #1

Inline Assembly

- Compiler will try to optimize code as much as possible
- However, we may still need to optimize manually by giving the compiler some assistance
- Use "__inline" keyword to notate a certain function that should be placed in the assembly directly and not to be called as a subroutine (save branching and returning overhead)
- Furthermore, some functions can also be written in assembly

Inline Assembly Syntax

- Invoke with __asm keyword anywhere a statement is expected
- Using either single line or multiple lines
- On single line
  - __asm("instruction[;instruction]"); //must be a single string
  - __asm {instruction[;instruction]}
  - Cannot include comments
  - Example: __asm("ADD r1, r0, 1") or __asm {ADD r1, r0,1}
- On multiple lines
  - __asm

  {

  …

  instruction

  …

  }
  - Can include comments anywhere

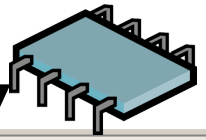
Restrictions On Using __asm

- The compiler optimize your codes, so the final codes may differ from what you wrote
- Cannot use all the ARM instructions, eg. BX and SVC instructions
- The compiler would not be aware if you change the mode
- Cannot change the program counter
- Should not modify the stack in any way
- Use registers r0-r3, sp, lr and the NZCV flags in CPSR with caution as other C expressions may corrupt them
- The following instructions are not supported
  - BX and SVC instructions
  - LDR Rn = expression pseudo-instruction
  - LDRT, LDRBT, STRT and STRBT instructions
  - MUL, MLA, UMULL, UMLAL, SMULL AND SMLAL flag setting instructions
  - MOV, MVN where 2nd operand is a constant
  - User mode LDM instructions
  - ADR and ADRL pseudo-instructions

Example 2: Inline assembly

```
• __inline int myadd (int x, int y)
{
    int result;
    __asm{ADD result, x, y};        //myadd is written in ARM
    return result;
}
• int main (void)
{  //main is written in C
int add2numbers, n1 = 3, n2 = 5;
add2numbers = myadd (n1, n2); //call inline asm function
}
```

Demo 2: Inline Assembly



Project

- Target 1
  - Source Group 1
    - Startup.s
    - 14InLineAsm.c
    - Retarget.c
    - Serial.c
    - Init_serial.c

14InLineAsm.c

```
01 #include <stdio.h>                    /* prototype declarations
02 #include <LPC21xx.H>                  /* LPC21xx definitions
03 extern void Init_serial(void);
04
05 __inline int myadd(int x, int y)
06 {
07     int result;
08
09     __asm{ADD result, x, y};
10
11     return result;
12 }
13
14 /*****************/
15 /* main program */
16 /*****************/
17 int main (void)  {                   /* execution starts here
18
19   int add2numbers, n1=3,n2=5;
20
21   Init_serial();                     /* initialize the serial i
22
23   add2numbers = myadd(n1, n2);
24
25   printf ("%d + %d = %d", n1, n2, add2numbers);
26
27   while (1) {                                /* An embedded progra
28     ;  /* ... */                             /* never returns. We
29   }                                          /* Replace the dots (
30 }
```

Embedded assembly

- For large subroutine
- Allows declaration of assembly functions in C with full functional prototypes, including arguments and return value
- Have overheads as a function
- Have access to full ARM and THUMB instruction sets

Embedded Assembly Syntax

- Functions declared with __asm can have arguments and return value

```
__asm return-type function-name (parameters list)
{
        …
        instruction
        …
}
```

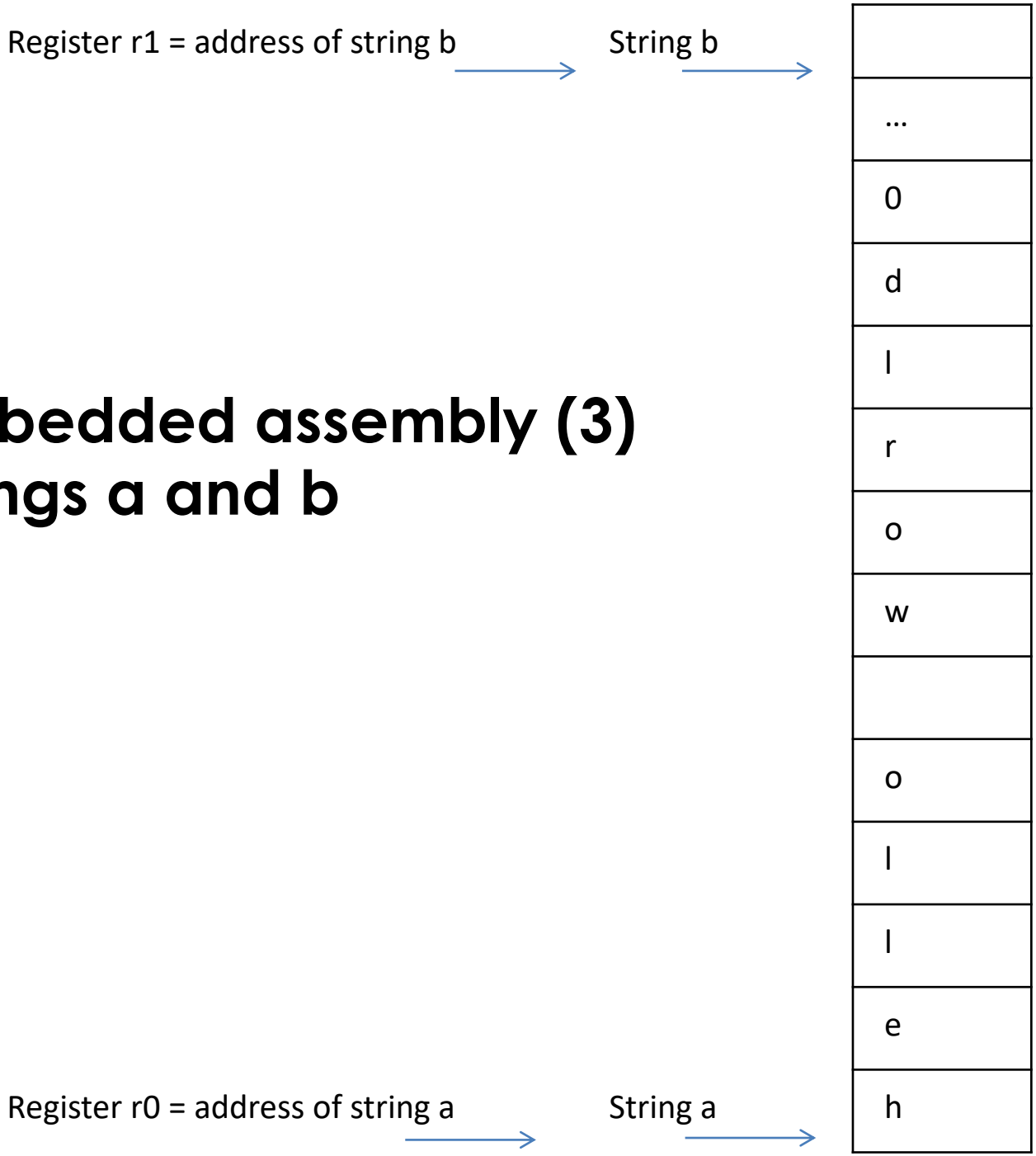
Example 3: Embedded Assembly (1)

```
#include <stdio.h>

extern void init_serial(void);              //initializes the serial driver

__asm void my_strcopy (const char *src, char *dst)
{
loop
    LDRB  r2, [r0], #1
    STRB  r2, [r1], #1
    CMP    r2, #0;check termination
    BNE    loop
    BX       lr
}
```

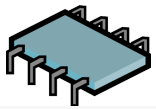
Example 3: Embedded Assembly (2)

```
int main(void)
{
    const char *a = “hello world”; //12 characters long
    char b[12];                    //array of 12 characters = string

    init_serial();

    my_strcopy(a, b);

    printf(“original string: ‘%s’\n”, a);
    printf(“Copied string: ‘%s’\n”, b);
    return 0;
}
```

Example 3: Embedded assembly (3) – strings a and b



Demo 3: Embedded Assembly



- Target 1
  - Source Group 1
    - Startup.s
    - 14EmbeddedAsm.c
    - Retarget.c
    - Serial.c
    - Init_serial.c

Project | Registers

```
04
05  __asm void myStrcpy(const char *scr, char *dst)
06  {
07  loop
08      LDRB r2, [r0], #1
09      STRB r2, [r1], #1
10      CMP  r2, #0
11      BNE  loop
12      BX   lr
13
14  }
15  /*****************/
16  /* main program */
17  /*****************/
18  int main (void)  {                  /*execution starts
19      const char *a="hello world";/*array of 12 chara
20      char b[12];
21      Init_serial();                  /*initialize the se
22      myStrcpy(a, b);
23      printf ("original string: %s\n", a);
24      printf ("copied string: %s\n", b);
```

UART #1

```
original string: hello world
copied string: hello world
```

Calling Between C And Assembly

- Functions can be written in C or assembly (store in separate files) and then mix together
- They can be called upon one another but must follow AAPCS standard and uses C calling conventions

C Directives - #Include And Extern

#include *file*

- Tells the preprocessor to treat the contents of a specified *file* as if those contents had appeared in the source program at the point where the directive appears.
- You can organize constant and macro definitions into include files and then use **#include** directives to add these definitions to any source file
- Include files are also useful for incorporating declarations of external variables and complex data types
- You need to define and name the types only once in an include file created for that purpose

extern

- Use the extern directive to declare global data and procedures as external
- Indicates to the compiler that a function is written in a different programming language

More Assembly Directives

PRESERVE8

- Specifies that the current file preserves eight-byte alignment of the stack
- LDRD and STRD instructions only work correctly if the address they access is eight-byte aligned

EXPORT *symbol*

- Use EXPORT to give code in other files access to *symbol* in the current file

IMPORT *symbol*

- Provides the assembler with a name that is not defined in the current assembly
- It is resolved at link time to a symbol defined in a separate object file
- The symbol is treated as a program address

Ex 4: Call Assembly Subroutine From C (1)

C code (caller)

```
#include <stdio.h>   /* prototype declarations for I/O functions  */
#include <LPC21xx.H>     /* LPC21xx definitions                    */
extern void Init_serial(void);
extern void revStr(const char *s, char *d);
```

Ex 4: Call Assembly Subroutine From C (2)

C code (caller)

```
/* main program */
int main (void)  {              /* execution starts here                */
        const char *src = "stressed";
        char dst[9];

        Init_serial();  /* initialize the serial interface          */

        revStr (src, dst);   /*call asm subroutine revStr              */
        printf ("%s when reads in reverse is %s\n", src, dst);

        while (1) {                 /* An embedded program does not stop and
        */
        ;  /* ... */            /* never returns. We use an endless loop.     */
        }                       /* Replace the dots (...) with your own code. */
}
```

Ex 4: Call Assembly Subroutine From C (3)

Assembly code (callee)

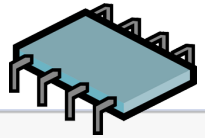
```
;input r0 points to src string "stressed"
;output r1 points to dst string
        PRESERVE8
        AREA reverseStr, CODE, READONLY
        EXPORT revStr
        ENTRY
revStr
        STMFD sp!, {r4-r5, lr}     ;save temporary registers

        ;get length of src
        MOV r4, #0                 ;loop counter - temporary
loop1
        LDRB r5, [r0], #1 ;get character - temporary
        CMP   r5, #0                          ;end of string?
        BEQ  rev
        ADD  r4, r4, #1            ;increment counter
        B    loop1
```

Ex 4: Call Assembly Subroutine From C (4)

Assembly code (callee)

```
rev             ;start reversing
    SUB  r0, r0, #1          ;adjust src pointer
loop2
    LDRB r5, [r0, #-1]!      ;get src character
    STRB r5, [r1], #1  ;store to dst
    SUBS r4, r4, #1
    BGT   loop2

    MOV   r5, #0
    STRB r5, [r1]        ;terminate dst string
    LDMFD sp!, {r4-r5, pc}     ;restore temporary registers
    END
```

Demo 4: Call Assembly Subroutine From C



Project

- Target 1
  - Source Group 1
    - Startup.s
    - 14callASMfromCmain.c
    - 14callASMfromCsub.s
    - Init_serial.c
    - Retarget.c
    - Serial.c

Project | Registers

14callASMfromCmain.c | 14callASMfromCsub.s

```
01 #include <stdio.h>                  /* prototype declarations fo
02 #include <LPC21xx.H>                /* LPC21xx definitions
03 extern void Init_serial(void);
04 extern void revStr(const char *s, char *d);
05
06 /* main program */
07 int main (void)  {                  /* execution starts here
08     const char *src = "stressed";
09     char dst[9];
10
11     Init_serial();                  /* initialize the serial int
12
13     revStr (src, dst);
14     printf ("%s when reads in reverse is %s\n", src, dst);
15
```

UART #1

```
stressed when reads in reverse is desserts
```

The ARM APCS (AAPCS)

- Application Procedure Call Standard → a standard
- Defines how subroutines can be separately written, separately compiled and separately assembled
- Contract between subroutine callers and callees
- Standard specifies
  - how parameters be passed to subroutines
  - which registers must have their content preserved (which are corruptible)
  - special roles for certain registers
  - a Full Descending stack pointed by r13 (sp)
  - etc

AAPCS Simplified Specifications

Register	Notes
r0 – r3	Parameters to and results from subroutines. Otherwise may be corrupted.
r4 – r11	Variables. Must be preserved.
r12	Scratch register (corruptible)
r13	Stack pointer (sp)
r14	Link register (lr)
r15	Program counter (pc)