

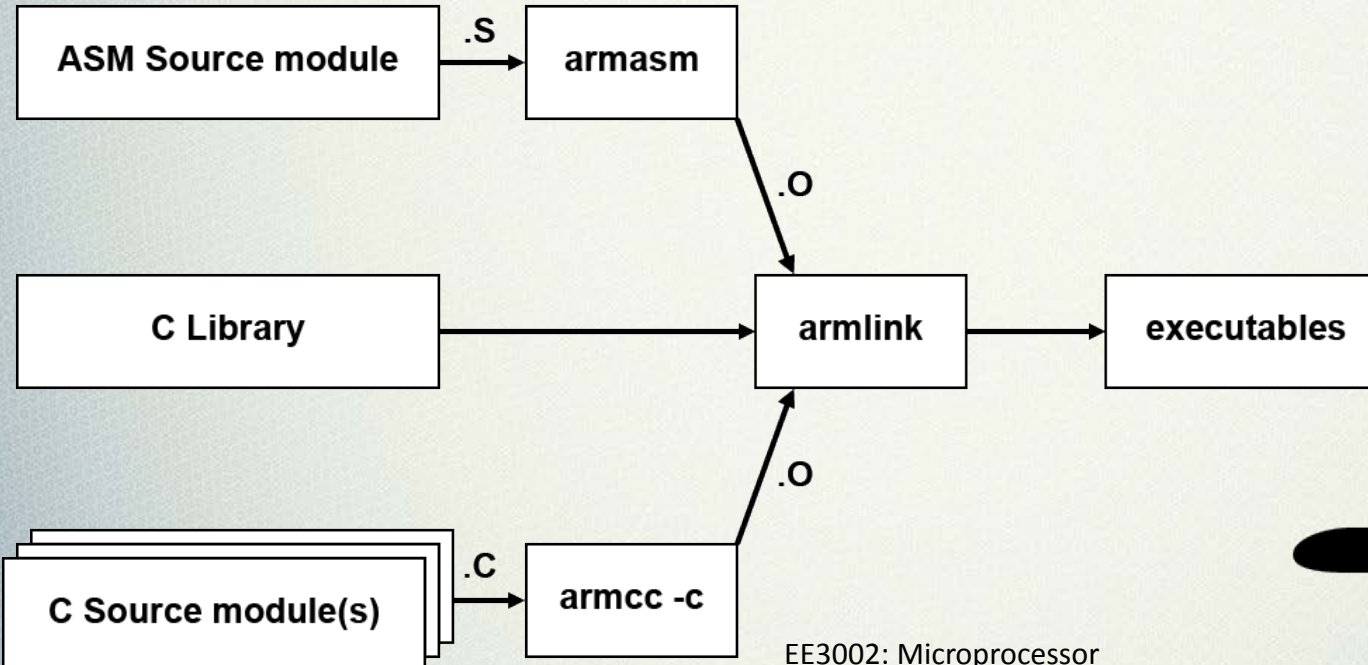
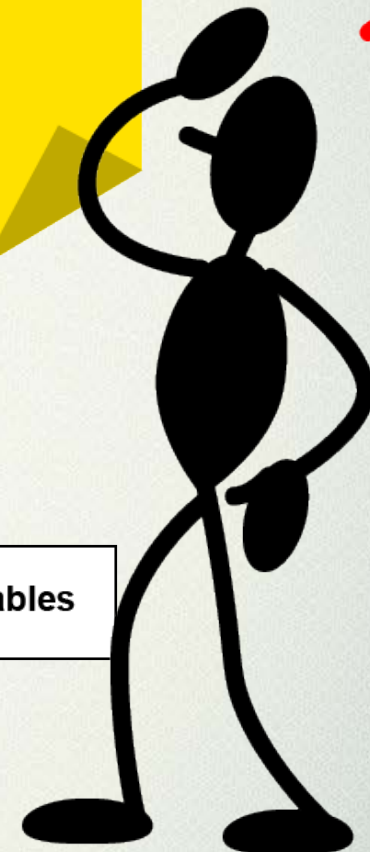
C &
Assembly

EE3002/IM2002
Microprocessor
Part 2

Mixing C and Assembly (Chapter 14)

~~Excuse me, could you please make way?~~

Siam lah!



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

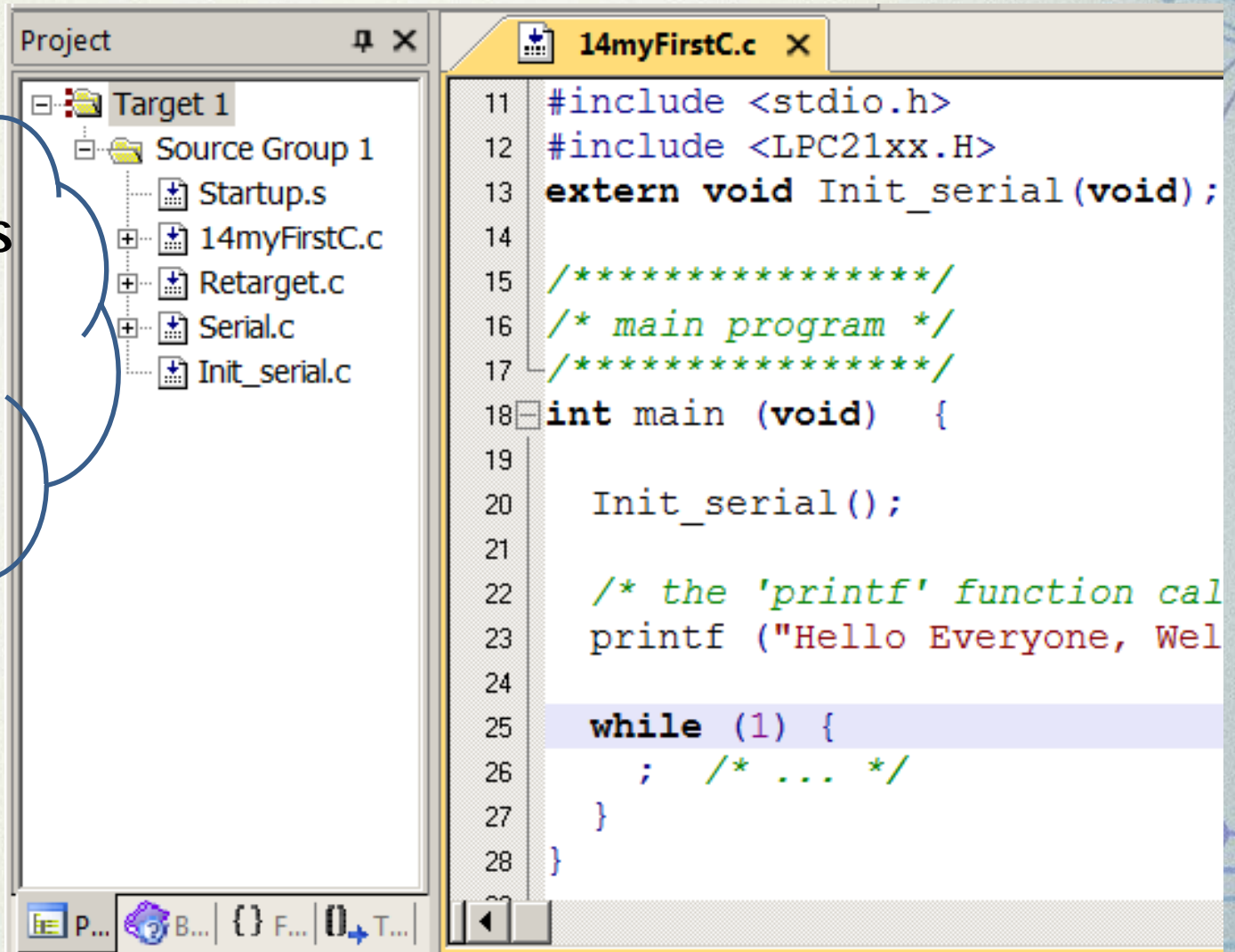


efficient

C Programming in ARM

Requirements

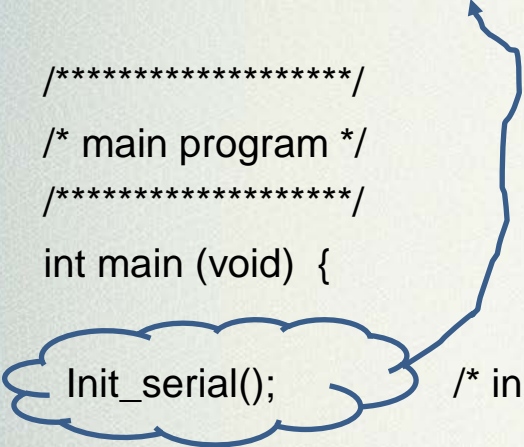
1. Startup.s
2. Retarget.c
3. Serial.c
4. Init_serial.c



Example 1: Myfirstc.C

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

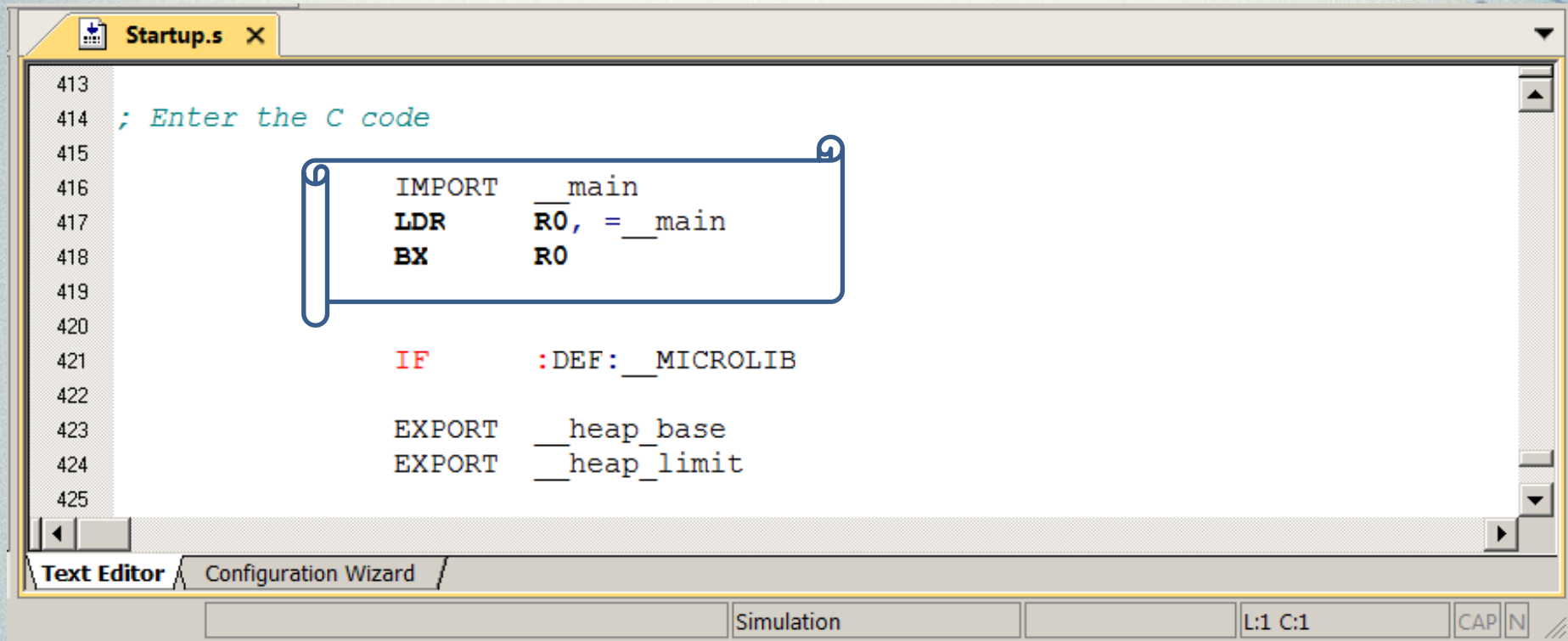
*/`
 - `/* main program */`
 - `/*

*/`
 - `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. */`
 - `}`
- 

Startup.s (1)

```
Startup.s X
230          AREA      RESET, CODE, READONLY
231          ARM
232
233
234 ; Exception Vectors
235 ; Mapped to Address 0.
236 ; Absolute addressing mode must be used.
237 ; Dummy Handlers are implemented as infinite loops which can be modified.
238
239 Vectors      LDR      PC, Reset_Addr
240              LDR      PC, Undef_Addr
241              LDR      PC, SWI_Addr
242              LDR      PC, PAbt_Addr
243              LDR      PC, DAbt_Addr
244              NOP                               ; Reserved Vector
245 ;          LDR      PC, IRQ_Addr
246              LDR      PC, [PC, #-0xFF0]         ; Vector from VicVectAddr
247              LDR      PC, FIQ_Addr
```

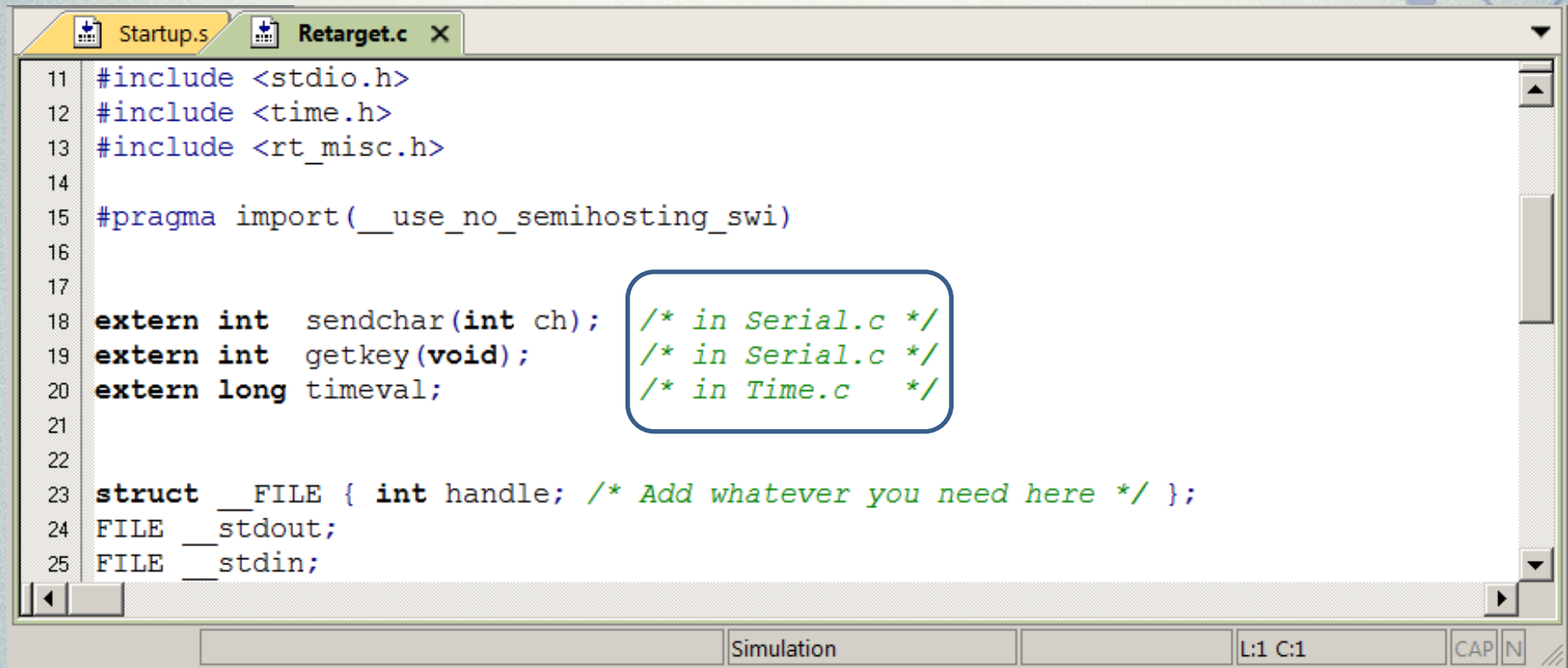
Startup.s (2)



```
413
414 ; Enter the C code
415
416     IMPORT    __main
417     LDR       R0, =__main
418     BX       R0
419
420
421     IF      :DEF:__MICROLIB
422
423     EXPORT   __heap_base
424     EXPORT   __heap_limit
425
```

Text Editor | Configuration Wizard | Simulation | L:1 C:1 | CAP | N

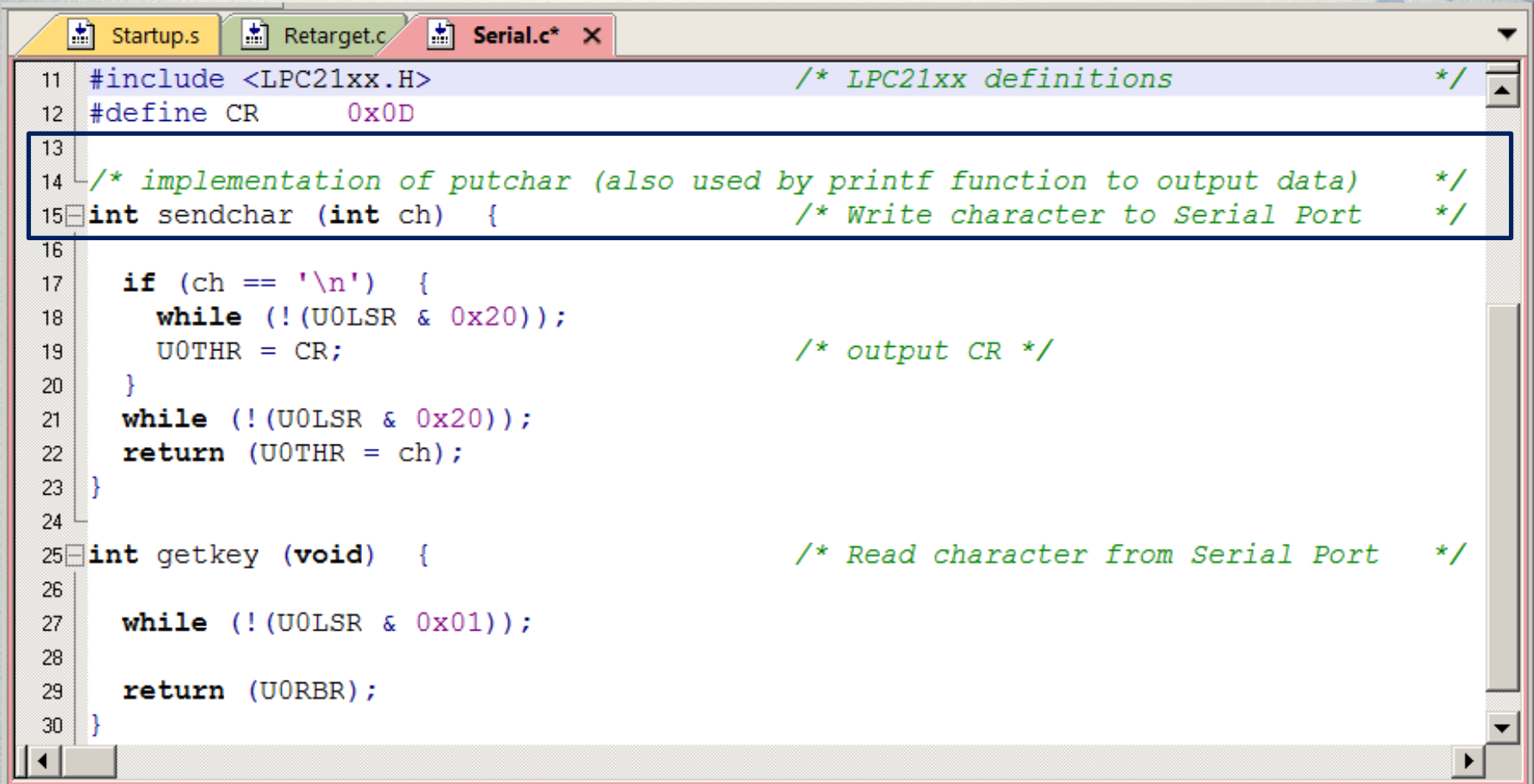
Retarget.c



```
11 #include <stdio.h>
12 #include <time.h>
13 #include <rt_misc.h>
14
15 #pragma import(__use_no_semihosting_swi)
16
17
18 extern int  sendchar(int ch); /* in Serial.c */
19 extern int  getkey(void);    /* in Serial.c */
20 extern long timeval;        /* in Time.c */
21
22
23 struct __FILE { int handle; /* Add whatever you need here */ };
24 FILE __stdout;
25 FILE __stdin;
```

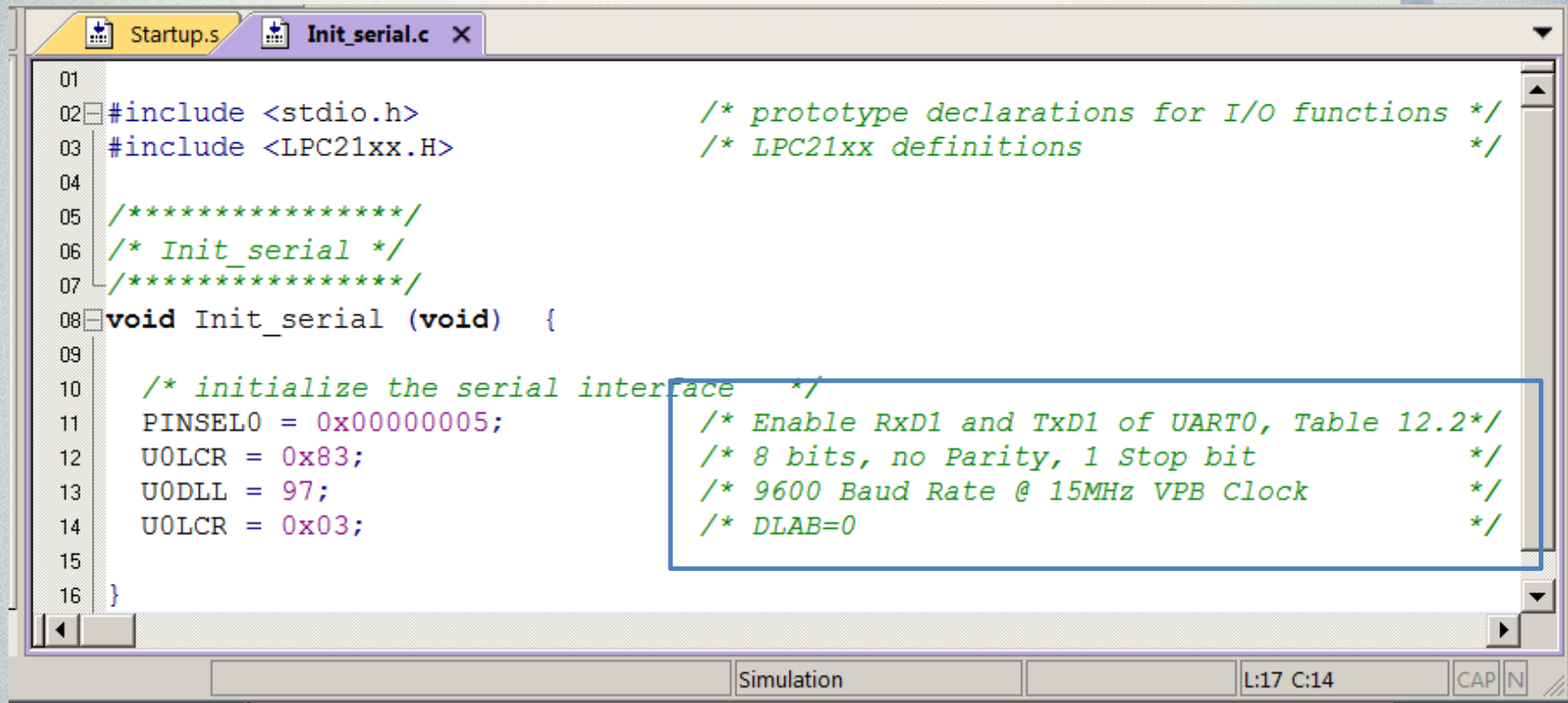
Simulation L:1 C:1 CAP N

Serial.c



```
11 #include <LPC21xx.H>                                /* LPC21xx definitions */
12 #define CR      0x0D
13
14 /* implementation of putchar (also used by printf function to output data) */
15 int sendchar (int ch) {                               /* Write character to Serial Port */
16
17     if (ch == '\n') {
18         while (!(U0LSR & 0x20));
19         U0THR = CR;                                   /* output CR */
20     }
21     while (!(U0LSR & 0x20));
22     return (U0THR = ch);
23 }
24
25 int getkey (void) {                                   /* Read character from Serial Port */
26
27     while (!(U0LSR & 0x01));
28
29     return (U0RBR);
30 }
```

Init_serial.c



```
01
02 #include <stdio.h>                                /* prototype declarations for I/O functions */
03 #include <LPC21xx.H>                                /* LPC21xx definitions */
04
05 /***/
06 /* Init_serial */
07 /***/
08 void Init_serial (void) {
09
10     /* initialize the serial interface */
11     PINSEL0 = 0x00000005;    /* Enable RxD1 and TxD1 of UART0, Table 12.2*/
12     UOLCR = 0x83;           /* 8 bits, no Parity, 1 Stop bit */
13     UODLL = 97;             /* 9600 Baud Rate @ 15MHz VPB Clock */
14     UOLCR = 0x03;           /* DLAB=0 */
15
16 }
```

Simulation L:17 C:14 CAP N

View output in Serial Windows → UART1 window

The screenshot shows the Keil uVision IDE interface. The 'View' menu is open, and 'Serial Windows' is selected. The sub-menu shows 'UART #1' as the active selection. A blue arrow points from the text 'Retarget to UART #1' to the 'UART #1' option. The background shows assembly and C code.

Assembly code snippet:

```
0134 E28F0008 ADD R0,PC,#0x00000008
```

C code snippet:

```
end user licence from KEIL for a compatible version
development tools. Nothing else gives you the right
*****

#include <stdio.h> /* prototype declara
#include <LPC21xx.H> /* LPC21xx definitio
tern void Init_serial(void);

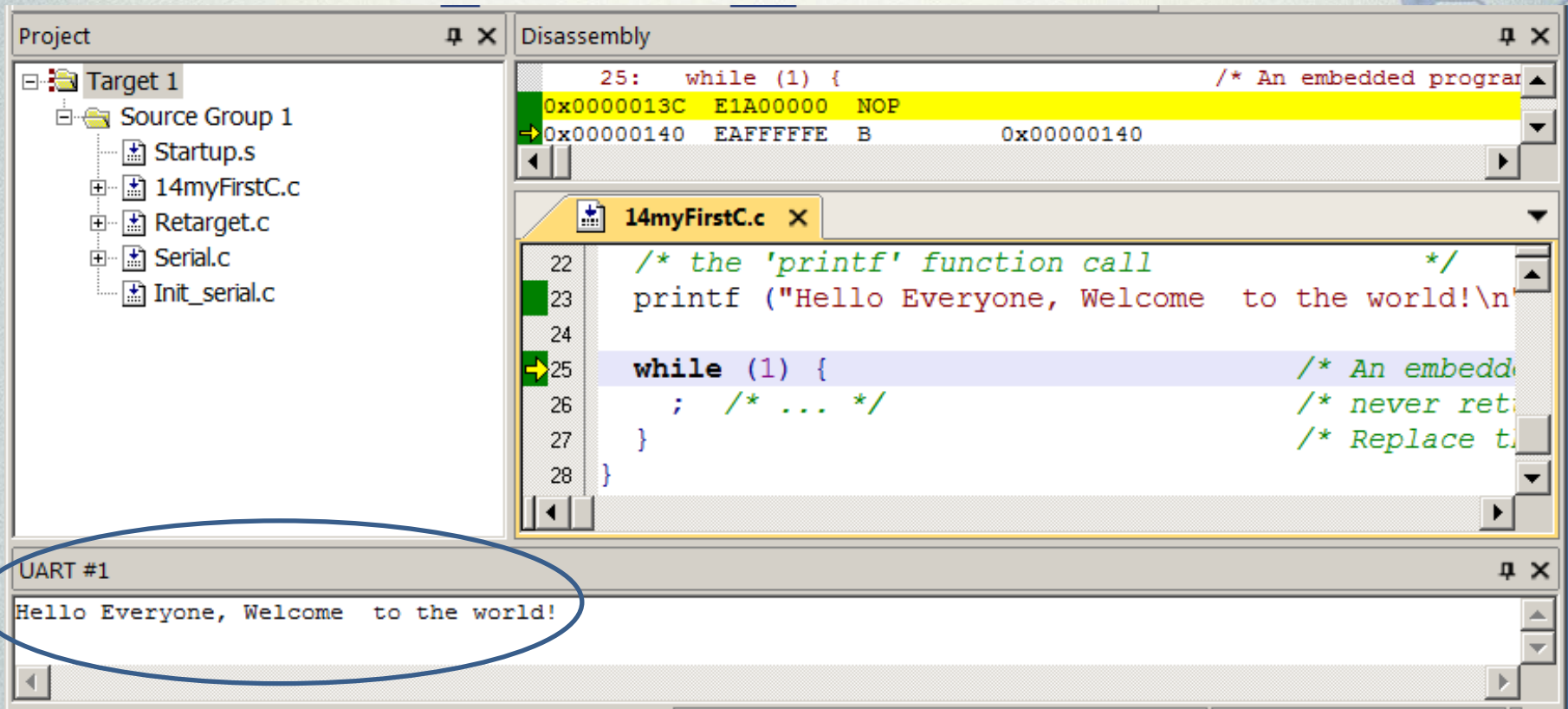
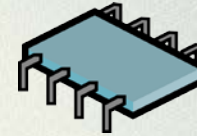
*****/

/* execution starts
/* initialize the se

Init_serial(),

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

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

Rules For Using __Asm

- Use ";" to separate instruction for multiple instructions in a single line
- If you use double quotes, enclose with single quote for all the instructions within the double quotes
- Use backslash character "\" to continue an instruction into the next line
- Use comments only in multiple lines format
- The comma "," is used as separator in the instruction
- Register names are treated as C variables and do not necessarily correspond to the physical registers
- Do not save and restore registers in inline assembler. The compiler will do it for you

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



The screenshot shows a code editor window with a project tree on the left and a code editor on the right. The project tree shows a 'Target 1' folder containing a 'Source Group 1' with files: 'Startup.s', '14InLineAsm.c', 'Retarget.c', 'Serial.c', and 'Init_serial.c'. The code editor displays the contents of '14InLineAsm.c'. The code includes standard headers, a serial initialization function, and a custom 'myadd' function using inline assembly. The 'myadd' function is defined with the keyword 'inline' and uses the '__asm' directive to perform an addition. The 'main' function calls 'myadd' and prints the result. A 'while' loop is present at the bottom, intended to keep the program running.

```
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 program
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
    ...
}
```


Restrictions On Embedded Assembly

- No return instructions are generated by the compiler
- A return instruction must be included if you want return value

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;
}
```

Register r1 = address of string b

String b

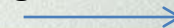
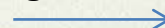


Example 3: Embedded assembly (3)

- strings a and b

Register r0 = address of string a

String a



Demo 3: Embedded Assembly



The screenshot shows an IDE with a project tree on the left and a code editor on the right. The project tree shows a 'Target 1' folder containing a 'Source Group 1' with files: 'Startup.s', '14EmbeddedAsm.c', 'Retarget.c', 'Serial.c', and 'Init_serial.c'. The code editor displays the assembly code for 'myStrncpy' and the C code for 'main'. The assembly code uses ARM instructions: LDRB, STRB, CMP, BNE, and BX. The C code includes comments and uses printf to output the original and copied strings. The UART #1 window at the bottom shows the output: 'original string: hello world' and 'copied string: hello world'.

```
04  
05 _asm void myStrncpy(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 s  
22     myStrncpy(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

14callASMfromCmain.c

```
01 #include <stdio.h> /* prototype declarations for */
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
```

14callASMfromCsub.s

UART #1

stressed when reads in reverse is desserts

Example 5: Call C Function From Asm (1)

```
#include <stdio.h>          /* prototype declarations for I/O functions */
#include <LPC21xx.H>         /* LPC21xx definitions */

extern void Init_serial(void);
extern int f(int i);

/* main program */
int main (void) {           /* execution starts here */
    int i;
    Init_serial();          /* initialize the serial interface */

    i = 2;                  /*value of I goes into r0
                           */
```

- Passing parameters to a C function from an assembly program

return-type function-name (parameter list) { ... }

- registers r0 – r3 for first 4 parameters in the parameter list
- pop from the stack for 5th parameters onwards
- return value in register r0

Example 5: Call C Function From Asm (2)

```
/* call f(i) and then print result, result is in r0 */
```

```
/*call arm subroutine */
```

```
printf (" for i = %d,  $i+2i+3i+4i+5i = %d$ \n", i, f(i));
```

```
while (1) {          /* An embedded program does not stop and  
*/
```

```
; /* ... */          /* never returns. We use an endless loop.  
*/
```

```
}                    /* Replace the dots (...) with your own code.  
*/
```

```
}
```

Example 5: Call C Function From Asm (3)

```
;int f(int i) {return g(i, 2*i, 3*i, 4*i, 5*i);}
```

```
;i is in r0
```

```
PRESERVE8
```

```
EXPORT f
```

```
IMPORT g
```

```
AREA funcf, CODE, READONLY
```

```
ENTRY
```

```
f
```

```
STMFD sp!, {r4, lr} ;save r4
```

```
ADD    r1, r0, r0 ;2*i
```

```
ADD    r2, r1, r0 ;3*i
```

```
ADD    r3, r1, r1 ;4*i
```

```
ADD    r4, r1, r2 ;5*i
```

```
STMFD sp!, {r4} ;5th parameter is in stack
```

```
BL     g ;call function g
```

```
LDMFD sp!, {r4} ;remove 5th parameter from stack
```

```
LDMFD sp!, {r4, pc} ;restore r4
```

```
END
```


Example 5: Call C Function From Asm (4)

```
#include <stdio.h> /* prototype declarations for I/O functions */
#include <LPC21xx.H> /*LPC21xx definitions */

/* g function */
int g (int a, int b, int c, int d, int e)
{
    return (a + b + c + d + e);
}
```

Demo 5: Call C Function From asm (1)



The screenshot shows an IDE window with a project explorer on the left and a code editor on the right. The project explorer shows a project named 'Target 1' with a 'Source Group 1' containing files: 'Startup.s', '14callCfromAsmMain.c', '14callCfromAsmf.s', '14callCfromAsmg.c', 'Init_serial.c', 'Retarget.c', and 'Serial.c'. The code editor shows the contents of '14callCfromAsmMain.c'.

```
01 #include <stdio.h> /* prototype declarations f
02 #include <LPC21xx.H> /* LPC21xx definitions
03 extern void Init_serial(void);
04 extern int f(int i);
05
06 /* main program */
07 int main (void) { /* execution starts here
08     int i;
09     Init_serial(); /* initialize the serial in
10
11     i = 2; /*valueof i goes into r0
12
13     /*call f(i) and then print result
14     printf ("for i = %d, i+2i+3i+4i+5i = %d\n",i, f(i));
15
```


Demo 5: Call C Function From asm (2)

The screenshot displays a debugger interface with two main panels. The left panel shows the 'Registers' window, and the right panel shows the 'Disassembly' window.

Registers Window:

Register	Value
R0	0x00000002
R1	0xE000C000
R2	0x40000068
R3	0x40000068
R4	0x00000002
R5	0x40000008
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000610
R11	0x00000000
R12	0x00000144
R13 (SP)	0x40000468
R14 (LR)	0x00000154
R15 (PC)	0x000005AC
CPSR	0x600000D3
SPSR	0x00000000
User/System	

Disassembly Window:

The disassembly window shows the assembly code for the function `f`. The current instruction is highlighted in yellow:

```
9: STMFD sp!, {r4, lr}
0x000005AC E92D4010 STMDB R13!, {R4, R14}
10: ADD r1, r0, r0 ;2*i
```

The assembly code for the function `f` is as follows:

```
01 ;int f(int i) {return g(i, 2*i, 3*i, 4*i, 5*i);}
02 ;i is in r0
03 PRESERVE8
04 EXPORT f
05 IMPORT g
06 AREA funcf, CODE, READONLY
07 ENTRY
08 f
09 STMFD sp!, {r4, lr}
10 ADD r1, r0, r0 ;2*i
11 ADD r2, r1, r0 ;3*i
12 ADD r3, r1, r1 ;4*i
13 ADD r4, r1, r2 ;5*i
14 STMFD sp!, {r4} ;5th parameter is in stack
```

i = 2 pass to asm through register r0

Demo 5: Call C Function From asm (3)

Registers

Register	Value
R0	0x00000002
R1	0x00000004
R2	0x00000006
R3	0x00000008
R4	0x0000000A
R5	0x40000008
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000610
R11	0x00000000
R12	0x00000144
R13 (SP)	0x4000045C
R14 (LR)	0x00000154
R15 (PC)	0x000005C4
CPSR	0x600000D3

Disassembly

```
15:      BL      g      ;call function g
0x000005C4 EBFFFEF1 BL      g(0x00000190)
16:      LDMFD   sp!, {r4} ;remove 5th parameter from stack
```

Assembly

```
08 f
09      STMFD   sp!, {r4, lr}
10      ADD     r1, r0, r0      ;2*i
11      ADD     r2, r1, r0      ;3*i
12      ADD     r3, r1, r1      ;4*i
13      ADD     r4, r1, r2      ;5*i
14      STMFD   sp!, {r4}      ;5th parameter is in stack
15      BL      g              ;call function g
16      LDMFD   sp!, {r4}      ;remove 5th parameter from stack
17      LDMFD   sp!, {r4, pc}
18      END
19
```

Memory

Address: 0x4000045C

0x4000045C: 0A 00 00 00 02 00 00 00 54 01 00

Real-Time Agent: Target Stopped | Simulation | t1: 0.0000

Input parameters to C function g (a, b, c, d, e)

Demo 5: Call C Function From asm (4)

The screenshot displays a debugger window with three main panes. The top-left pane shows the 'Registers' window with a list of registers R0 through R11 and their current values. The top-right pane shows the 'Assembly' window with two instructions highlighted: `0x000001A0 E0800002 ADD R0,R0,R2` and `0x000001A4 E0800003 ADD R0,R0,R3`. The bottom pane shows the 'Locals' window, which is highlighted with a red rectangle. It contains a table of local variables a, b, c, d, and e with their respective values. The status bar at the bottom indicates 'Real-Time Agent: Target Stopped' and 'Simulation'.

Register	Value
R0	0x00000006
R1	0x00000004
R2	0x00000006
R3	0x00000008
R4	0x0000000A
R5	0x40000008
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000610
R11	0x00000000

```
#include <stdio.h> /* prototype declaration */
#include <LPC21xx.H> /* LPC21xx definitions */

/* g function */
int g (int a, int b, int c, int d, int e)
{
    return (a + b + c + d + e);
}
```

Name	Value
a	0x00000002
b	0x00000004
c	0x00000006
d	0x00000008
e	0x0000000A

Real-Time Agent: Target Stopped Simulation

Demo 5: Call C Function From asm (5)

The screenshot shows a debugger interface with two main panes: **Registers** and **Disassembly**.

Registers Pane:

Register	Value
R0	0x0000001E
R1	0x00000004
R2	0x00000006
R3	0x00000008
R4	0x0000000A
R5	0x40000008
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000610
R11	0x00000000
R12	0x00000002
R13 (SP)	0x40000460
R14 (LR)	0x000005C8
R15 (PC)	0x000005CC
CPSR	0x600000D3

Disassembly Pane:

17: LDMFD sp!, {r4, pc}
0x000005CC E8BD8010 LDMIA R13!, {R4, PC}
ARM common call via r1:

14callCfromAsmg.c 14callCfromAsmMain.c 14callCfromAsmf.s

```
08 f
09     STMFD sp!, {r4, lr}
10     ADD    r1, r0, r0    ;2*i
11     ADD    r2, r1, r0    ;3*i
12     ADD    r3, r1, r1    ;4*i
13     ADD    r4, r1, r2    ;5*i
14     STMFD sp!, {r4}      ;5th parameter is in stack
15     BL     g             ;call function g
16     LDMFD sp!, {r4}      ;remove 5th parameter from stack
17     LDMFD sp!, {r4, pc}
18     END
19
```

A red arrow points from the text "Return from C function g and result is in register r0" to the value of register R0 (0x0000001E).

Return from C function g and result is in register r0

Demo 5: Call C Function From asm (6)

The screenshot displays a debugger interface with three main panels:

- Registers:** A table showing the current state of registers R0 through R12. R0 is 0xFFFFFFFF, R1 is 0x40000000, R2 is 0x00000190, R3 is 0x00000168, R4 is 0x00000002, R5 is 0x0000001E, R6 is 0x00000000, R7 is 0x00000000, R8 is 0x00000000, R9 is 0x00000000, R10 is 0x00000610, R11 is 0x00000000, and R12 is 0x00000365.
- Disassembly:** Shows assembly instructions at memory addresses 0x00000168 and 0x0000016C. The instruction at 0x00000168 is `E1A00000 NOP`. The instruction at 0x0000016C is `EAFFFFF B 0x0000016C`.
- Source Code:** Displays the C source file `14callCfromAsmMain.c`. The code includes a `while (1) {` loop. A comment indicates that the value of `i` goes into `r0`. A `printf` statement prints the result of the calculation `i+2i+3i+4i+5i`. The `while` loop is currently executing.

At the bottom, the **UART #1** window shows the output of the program: `for i = 2, i+2i+3i+4i+5i = 30`. The word **result** is highlighted in red.

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)

Summary

- Why mix C with assembly?
- C programming in ARM
- Inline and embedded
- Call assembly from C function
- Call C function from assembly
- Passing of parameters between them - AAPCS