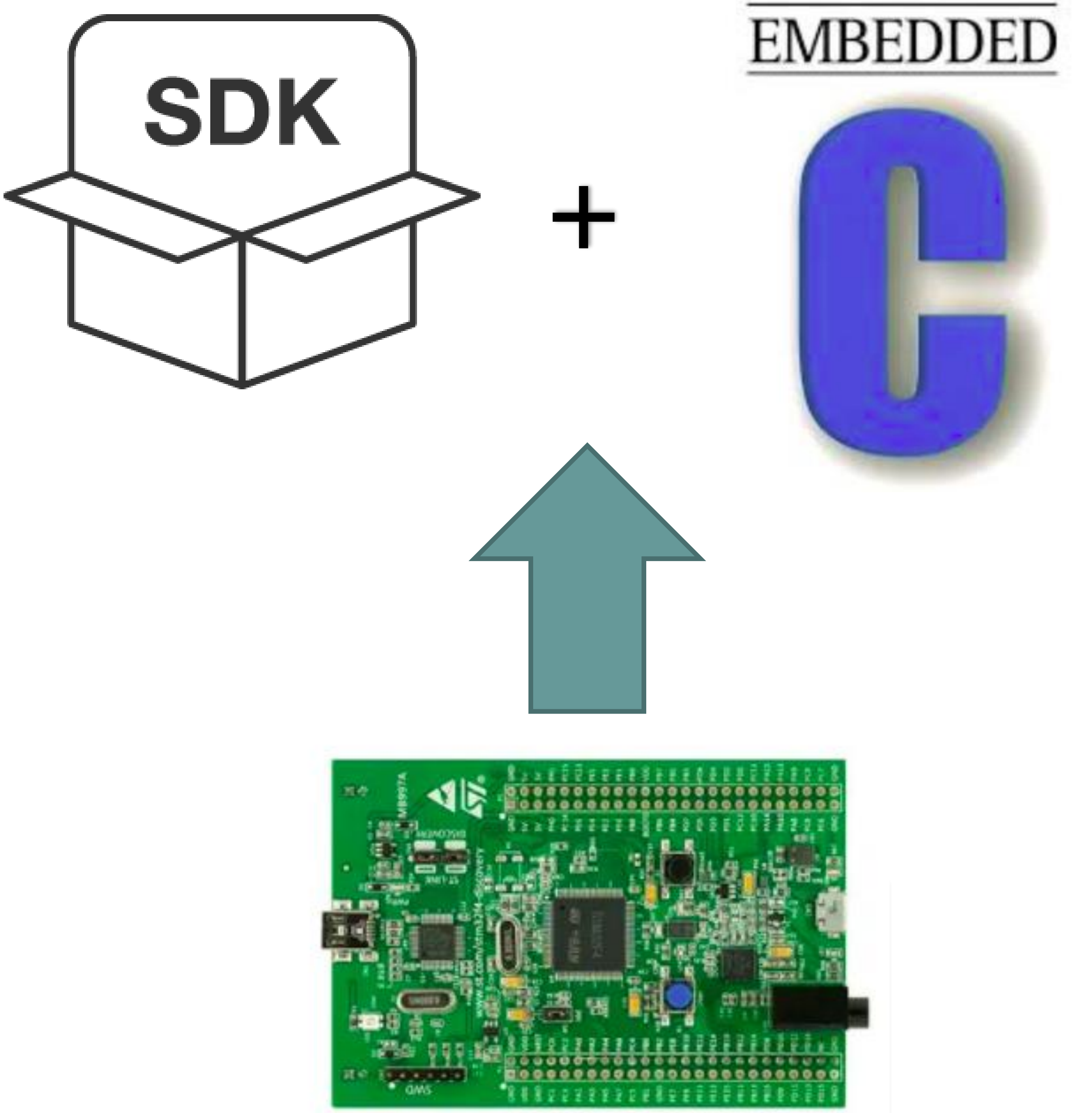
Hi-Tech Education Center

STM32 Basic
Training Course

HCM - 2021


SDK
+
EMBEDDED
C

INSTRUCTOR

TRƯƠNG NGỌC PHÚ (1991)

Embedded System Specialist

- Experience **8+** years Embedded Hardware & Software
- Have worked for: Tekbox, Renesas, BanVien MimosaTEK
- Expertise:
  - Embedded Hardware & Software
  - STM32 MPU, STM32 MCU
  - NXP, TI MCU
  - Embedded Linux
  - M2M & IoT system
  - Hardware Security
- Own Business: Smart Device & IoT Development

  http://kienminh.net/

Agenda

- Embedded System Overview
- Embedded C
- Keil-C Simulator
- Sample Project

Training Repo

- Training material - Github

https://github.com/kmtekvn/hitech_stm32_basic

- Git management tool

https://desktop.github.com/

- Software

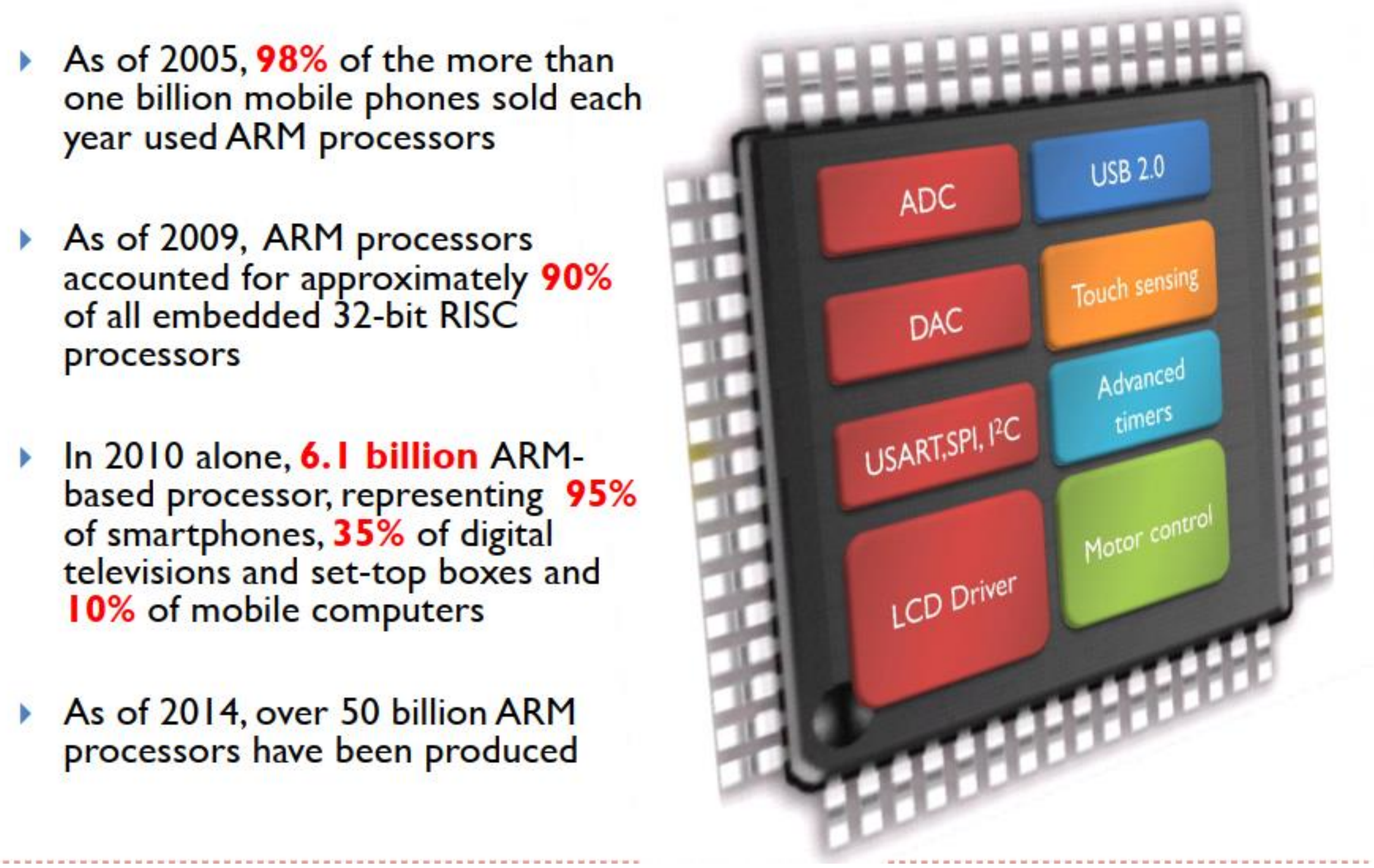
https://1drv.ms/u/s!AjRP3z7340A0xy4qGKA6xQHUkSGl?e=mhSbex

Embedded Overview

Embedded System

http://news.thomasnet.com/IMT/wp-content/uploads/sites/2/2014/08/google-glass.jpg

Why ARM Processor

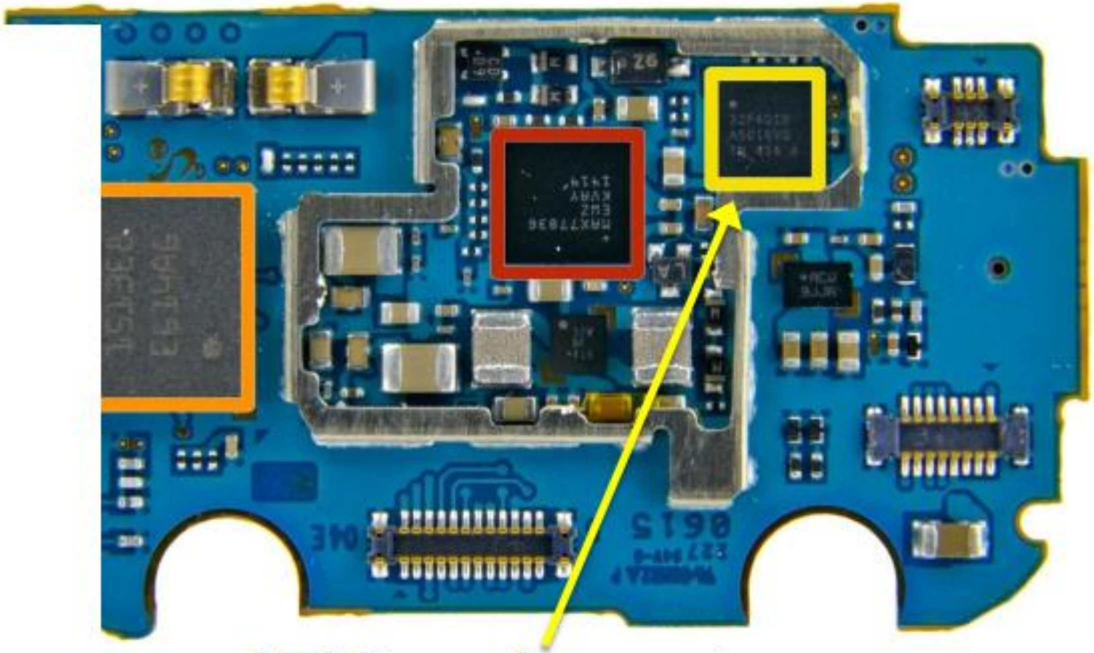


- As of 2005, **98%** of the more than one billion mobile phones sold each year used ARM processors
- As of 2009, ARM processors accounted for approximately **90%** of all embedded 32-bit RISC processors
- In 2010 alone, **6.1 billion** ARM-based processor, representing **95%** of smartphones, **35%** of digital televisions and set-top boxes and **10%** of mobile computers
- As of 2014, over 50 billion ARM processors have been produced

iPhone 5 Teardown

The A6 processor is the first Apple System-on-Chip (SoC) to use a custom design, based off the **ARMv7** instruction set.

Samsung Galaxy Gear



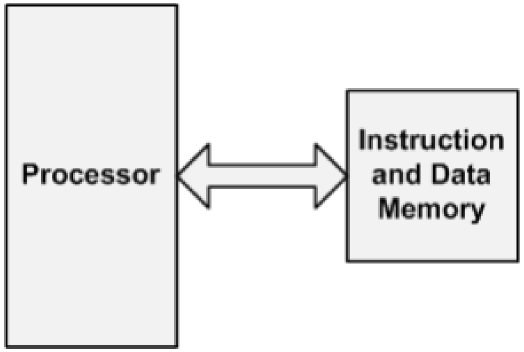
- STMicroelectronics STM32F401B **ARM-Cortex M4** MCU with 128KB Flash

source: *ifixit.com*

Computer Architecture

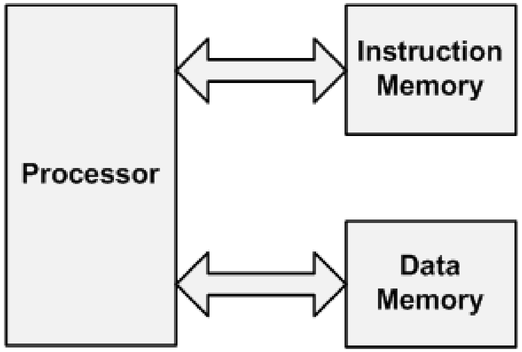
Von-Neumann

Instructions and data are stored in the same memory.



Harvard

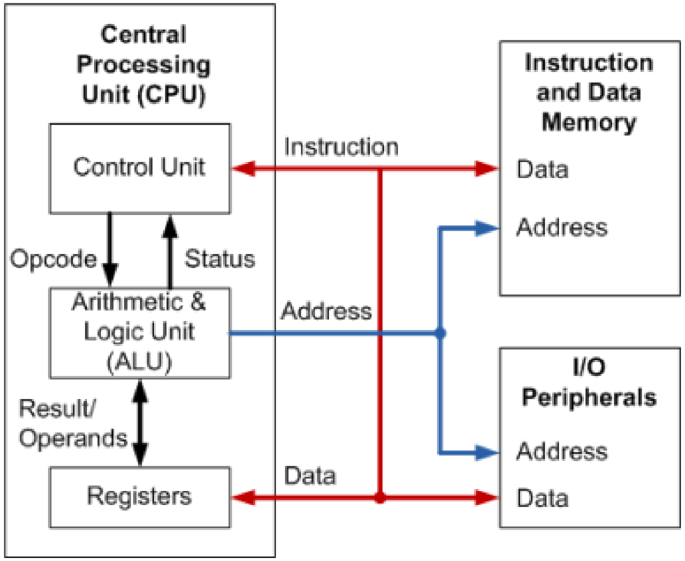
Data and instructions are stored into separate memories.



Computer Architecture

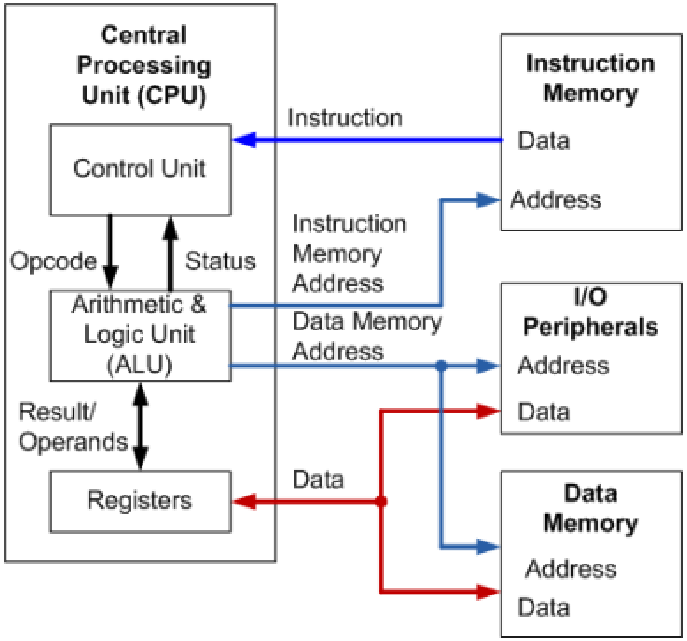
Von-Neumann

Instructions and data are stored in the same memory.



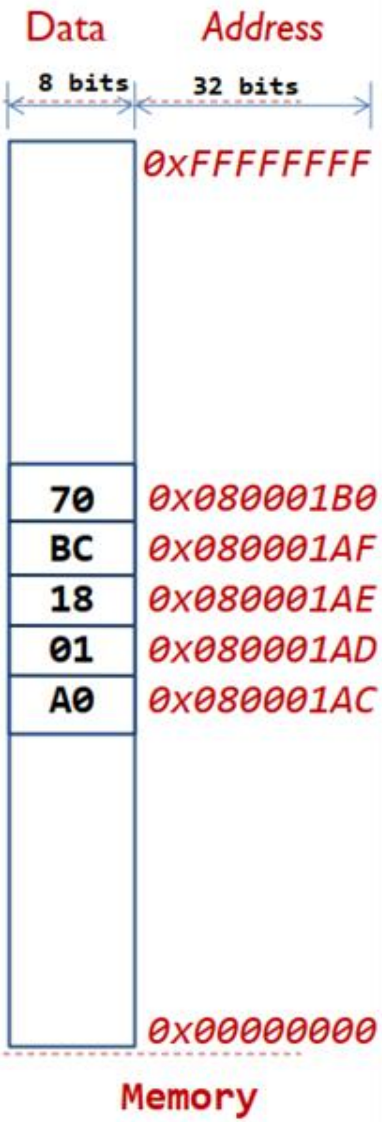
Harvard

Data and instructions are stored into separate memories.

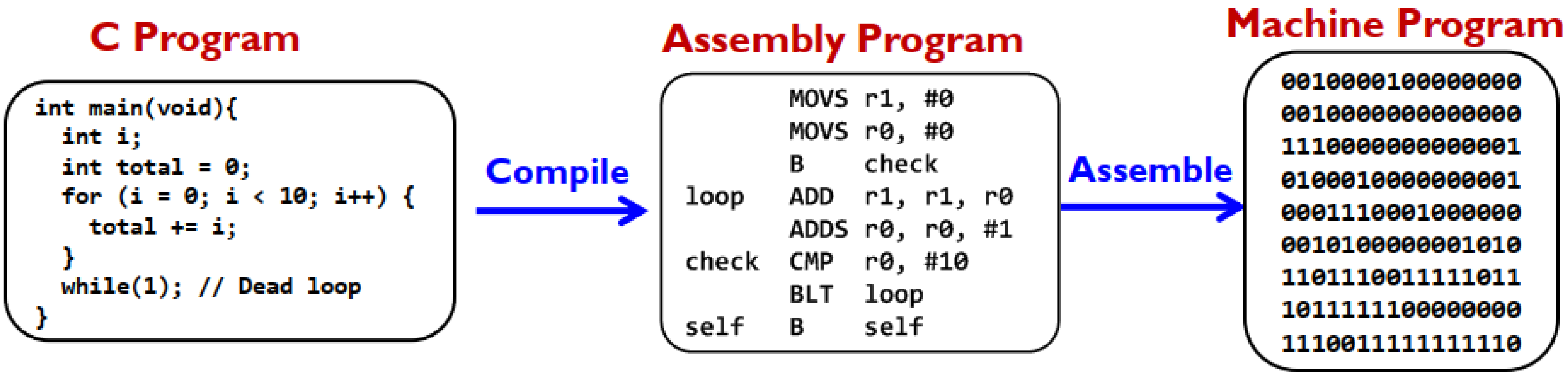


Memory

- Memory is arranged as a series of "locations"
  - Each location has a unique "address"
  - Each location holds a byte (***byte-addressable***)
  - e.g. the memory location at address `0x080001B0` contains the byte value `0x70`, i.e., 112
- The number of locations in memory is limited
  - e.g. 4 GB of RAM
  - 1 Gigabyte (GB) = 2^{30} bytes
  - 2^{32} locations ➔ 4,294,967,296 locations!
- Values stored at each location can represent either program data or program instructions
  - e.g. the value `0x70` might be the code used to tell the processor to add two values together



Levels of Program Code



```
int main(void){
  int i;
  int total = 0;
  for (i = 0; i < 10; i++) {
    total += i;
  }
  while(1); // Dead loop
}
```

```
        MOVS r1, #0
        MOVS r0, #0
        B    check
loop    ADD  r1, r1, r0
        ADDS r0, r0, #1
check   CMP  r0, #10
        BLT  loop
self    B    self
```

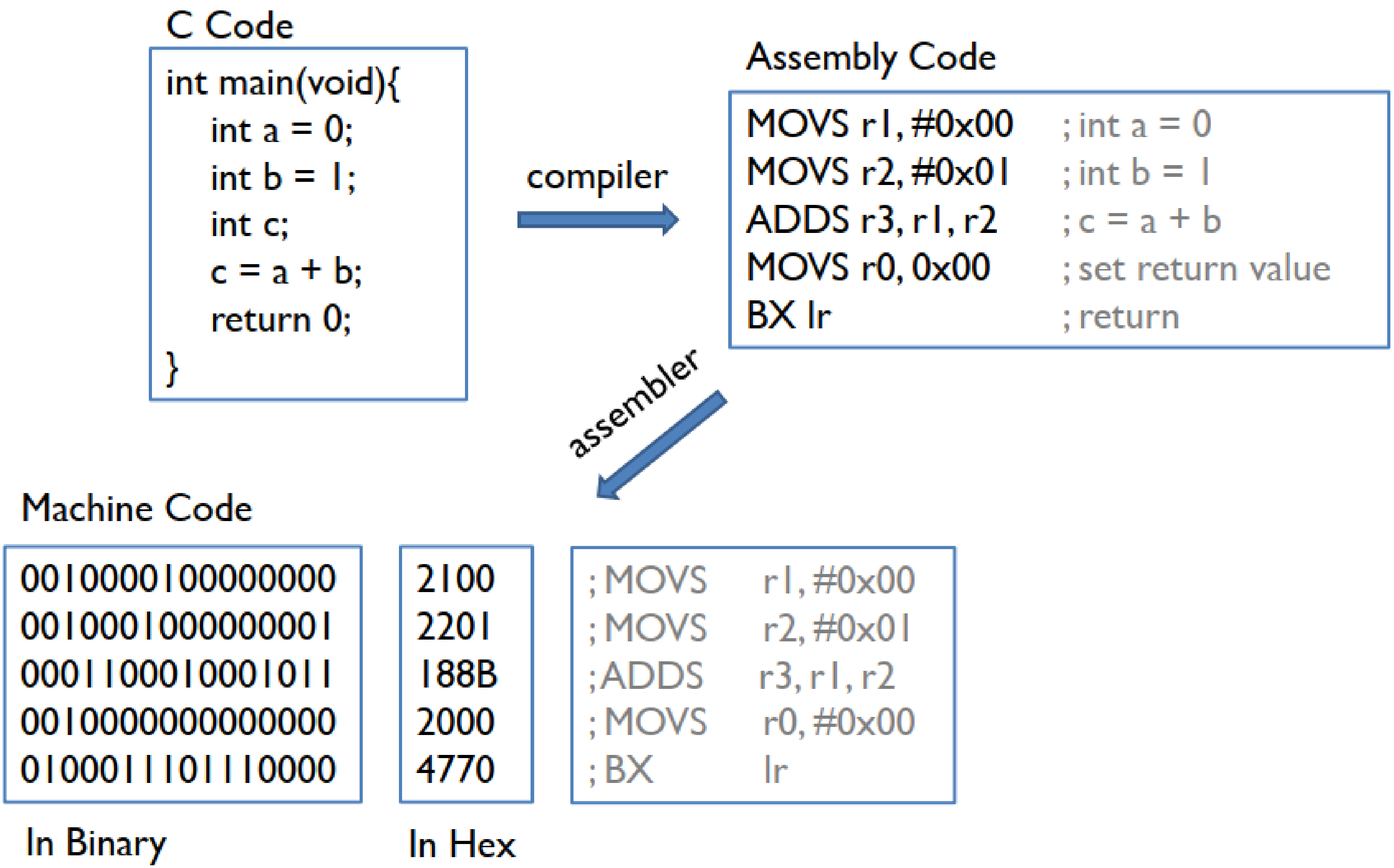
```
0010000100000000
0010000000000000
1110000000000001
0100010000000001
0001110001000000
0010100000001010
1101110011111011
1011111100000000
1110011111111110
```

- **High-level language**
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability

- **Assembly language**
  - Textual representation of instructions

- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data

See a Program Runs



C Code

```
int main(void){
    int a = 0;
    int b = 1;
    int c;
    c = a + b;
    return 0;
}
```

compiler

Assembly Code

```
MOVS r1, #0x00     ; int a = 0
MOVS r2, #0x01     ; int b = 1
ADDS r3, r1, r2    ; c = a + b
MOVS r0, 0x00      ; set return value
BX lr              ; return
```

assembler

Machine Code

```
0010000100000000
0010001000000001
0001100010001011
0010000000000000
0100011101110000
```

In Binary

```
2100
2201
188B
2000
4770
```

In Hex

```
; MOVS    r1, #0x00
; MOVS    r2, #0x01
;ADDS     r3, r1, r2
; MOVS    r0, #0x00
; BX      lr
```

Embedded C

Logical and Bitwise Operators

Logical Operators

- A logical operator is used to combine 2 or more conditions in an expression.
- Logical AND - &&
  - Operator **&&** returns true when both the conditions in consideration are true; else false
- Logical OR - ||
  - Operator **||** returns true when either or both the conditions in consideration are true; else false
- Logical NOT - !
  - Operator **!** returns true when either or both the conditions in consideration are true; else false
- Logical XOR
  - In the Boolean sense, this is just **!= (not equal)**

Logical example

```
int a = 10, b = 4, c = 10, d = 20;

// logical AND example
if (a > b && c == d)
    printf("a is greater than b AND c is equal to d\n");
    // doesn’t print because c != d

// logical OR example
if (a > b || c == d)
    printf("a is greater than b OR c is equal to d\n");
    // NOTE: because a>b, the clause c==d is not evaluated

// logical NOT example
if (!a)
    printf("a is zero\n");   // doesn’t print because a != 0
```

C Bitwise Operators

C has 6 operators for performing bitwise operations on integers

Operator	Meaning	
&	Bitwise AND	Result is 1 if both bits are 1
|	Bitwise OR	Result is 1 if either bit is 1
^	Bitwise XOR	Result is 1 if both bits are different
>>	Right shift	
<<	Left shift	
~	Ones complement	The logical invert, same as NOT

Bitwise Boolean examples

```
char j = 11;    // 0 0 0 0 1 0 1 1 = 11
char k = 14;    // 0 0 0 0 1 1 1 0 = 14

Bitwise Boolean Operators
char m = j & k; // 0 0 0 0 1 0 1 0 = 10
char n = j | k; // 0 0 0 0 1 1 1 1 = 15
char p = j ^ k; // 0 0 0 0 0 1 0 1 = 5

NOTE: This is a logical (not Boolean) operation
bool q = j && k;   // true == 1
bool q = 0 && k;   // false == 0
```

Shifting and Inversion

Shifting

Shifting

```
char j = 11;// 0 0 0 0 1 0 1 1 = 11
char k = j<<1;  // 0 0 0 1 0 1 1 0
= 22 (j*2)
char m = j>>1;  // 0 0 0 0 0 1 0 1
= 5  (j/2)
```

Shifting

```
char s1, s2, s3, s4;
s1=-11;         // 1 1 1 1 0 1 0 1 -11
s2=s1>>1;       // 1 1 1 1 1 0 1 0  -6

s3=117;         // 0 1 1 1 0 1 0 1  117
s4=s3>>1;       // 0 0 1 0 0 0 0 0  58
                // sign extension!

unsigned char u1, u2;
u1=255;         // 1 1 1 1 0 1 0 1  245
u2=u1>>1;       // 0 1 1 1 1 1 1 1  122
                // no sign extension!
```

Inversion

Logical invert

```
char j = 11;      // j = 0 0 0 0 1 0 1 1 =  11
char k = ~j;      // k = 1 1 1 1 0 1 0 0 = 244
                  // Note:         j + k = 255
```

Arrays and pointers

Array Identifiers & Pointers

```
char message_array[] = “Hello” ;
```

- Question: So what exactly is **message**?
- Answer: In C, an array name is a constant pointer that references the 0th element of the array's storage.
- **Constant** means it cannot be changed (just as we can't change the constant 3).

Consequences

- char message_array[] = “Hello” ;
- char *message = “Hello”;

Question: What is ***message**?

- *message == ‘H’; // an array pointer. It points to the
  // start of the array (to 0th element)

Read *message as “what message points to”

What is another expression for message?

- message == &message[0]; message[0]==‘H’

Pointer Variables and Arrays

```
char *hi = “Hello” ;
```

Allocates space and initializes a constant string “Hello”, then allocates space for pointer hi and initializes it to point to the 0th element.

```
char message[] = “Greetings!” ;
```

Allocates space for the array message and initializes its contents to the string “Greetings!”.

```
char *p_mesg = message ;
```

Allocates space for pointer p_mesg and initializes it to point to message.

```
char ch ;          // Declares ch as a char
p_mesg++ ;         // Advance p_mesg by one element (char in this case)
ch = *p_mesg ;     // Set ch to the character p_mesg points to (in this case 'r').
```

C Structures

C Structs

- A *struct* is a way of grouping named, heterogeneous data elements that represent a coherent concept.

C Structs

- A struct is a way of grouping named, heterogeneous data elements that represent a coherent concept.
- Example:

```
#define MAXNAME (20)

struct person {
    char name[MAXNAME+1] ;
    int age ;
    double income ;
} ;
```

C Structs

- Question: What is an object with no methods and only instance variables public?
- Answer: A struct! (well, sort of).
- A struct is a way of grouping named, heterogeneous data elements that represent a coherent concept.
- Example:

```
#define MAXNAME (20)

struct person {
   int age ;
   double income ;
} ;
```

coherent concept - the information recorded for a person.

C Structs

- Question: What is an object with no methods and only instance variables public?
- Answer: A struct! (well, sort of).
- A struct is a way of grouping named, heterogeneous data elements that represent a coherent concept.
- Example:

```
#define MAXNAME (20)

struct person {
    char name[MAXNAME+1];
    int age ;
    double income ;
} ;
```

Using Structs

- Declaration:

```
struct person {
    char name[MAXNAME+1] ; // explicit size known
     char *title;         // a pointer has explicit size
    char profession[];   // ILLEGAL, size not known
    int age ;
    double income ;
} ;
```

- Definitions:

```
struct person mike, pete, chris ;
```

- Assignment / field references ('dot' notation):

```
mike = pete ;// this does a shallow copy!!
   // If the structure contains pointers, the pointers will be
   // copied, but not what they point to.  Thus, after the copy,
   // there will be two pointers pointing to the same memory.
pete.age = chris.age + 3;
```

Using Structs

- Note: Space allocated for the whole struct at definition.
- Struct arguments are passed by value (i.e., copying)

WRONG

```
void give_raise( struct person p, double pct) {
    p.income *= (1 + pct/100) ;
    return ;
}

give_raise(mike, 10.0); // what is mike’s income after raise
```

RIGHT

```
struct person give_raise(struct person p, double pct) {
    p.income *= (1 + pct/100) ;
    return p ;
}

mike = give_raise(mike, 10.0) ; // what is mike’s income after raise?
```

Using Structs pointers

- Better if you can pass a pointer to the structure

```
void give_raise(struct person *p, double pct) {
    p->income *= (1 + pct/100) ;
    return ;
}

give_raise(&mike, 10.0) ;
```

Const qualifier

Const qualifier

- The const qualifier applied to a declared variable states the value cannot be modified.
- Using this feature can help prevent coding errors.
- Good for settings and configurations.

const char * - a pointer to a const char
the value being pointed to can't be changed but the pointer can.

char * const - is a constant pointer to a char
the value can be changed, but the pointer can‘t

Order can be confusing…

Const qualifier cont.

- To avoid confusion, always *append* the const qualifier.

```
int * mutable_pointer_to_mutable_int;

int const * mutable_pointer_to_constant_int;

int * const constant_pointer_to_mutable_int;

int const * const constant_ptr_to_constant_int;
```

Symbolic Names

typedef

Symbolic Type Names - typedef

- Suppose we have a pricing system that prices goods by weight.
  - Weight is in pounds, and is a double precision number.
  - Price is in dollars, and is a double precision number.
  - Goal: Clearly distinguish weight variables from price variables.

Symbolic Type Names - typedef

- Suppose we have a pricing system that prices goods by weight.
    - Weight is in pounds, and is a double precision number.
    - Price is in dollars, and is a double precision number.
    - Goal: Clearly distinguish weight variables from price variables.
- Typedef to the rescue:
    - typedef ***declaration*** ;

Creates a new "type" with the variable slot in the ***declaration***.

Symbolic Type Names - typedef

- Suppose we have a pricing system that prices goods by weight.
  - Weight is in pounds, and is a double precision number.
  - Price is in dollars, and is a double precision number.
  - Goal: Clearly distinguish weight variables from price variables.
- Typedef to the rescue:
  - typedef ***declaration*** ;Creates a new "type" with the variable slot in the ***declaration***.

Examples:

```
typedef double PRICE_t;  // alias for double to declare price variables
typedef double WEIGHT_t; // alias for double to declare weight variables
PRICE_t p ;              // double precision value that's a price
WEIGHT_t lbs ;           // double precision value that's a weight
```

typedef In Practice

- Symbolic names for array types

```
#define  MAXSTR  (100)

typedef  char  LONG_STRING_t[MAXSTR+1] ;

LONG_STRING_t line ;
LONG_STRING_t buffer ;
LONG_STRING_t *p_long_string;
```

typedef In Practice

- Symbolic names for array types

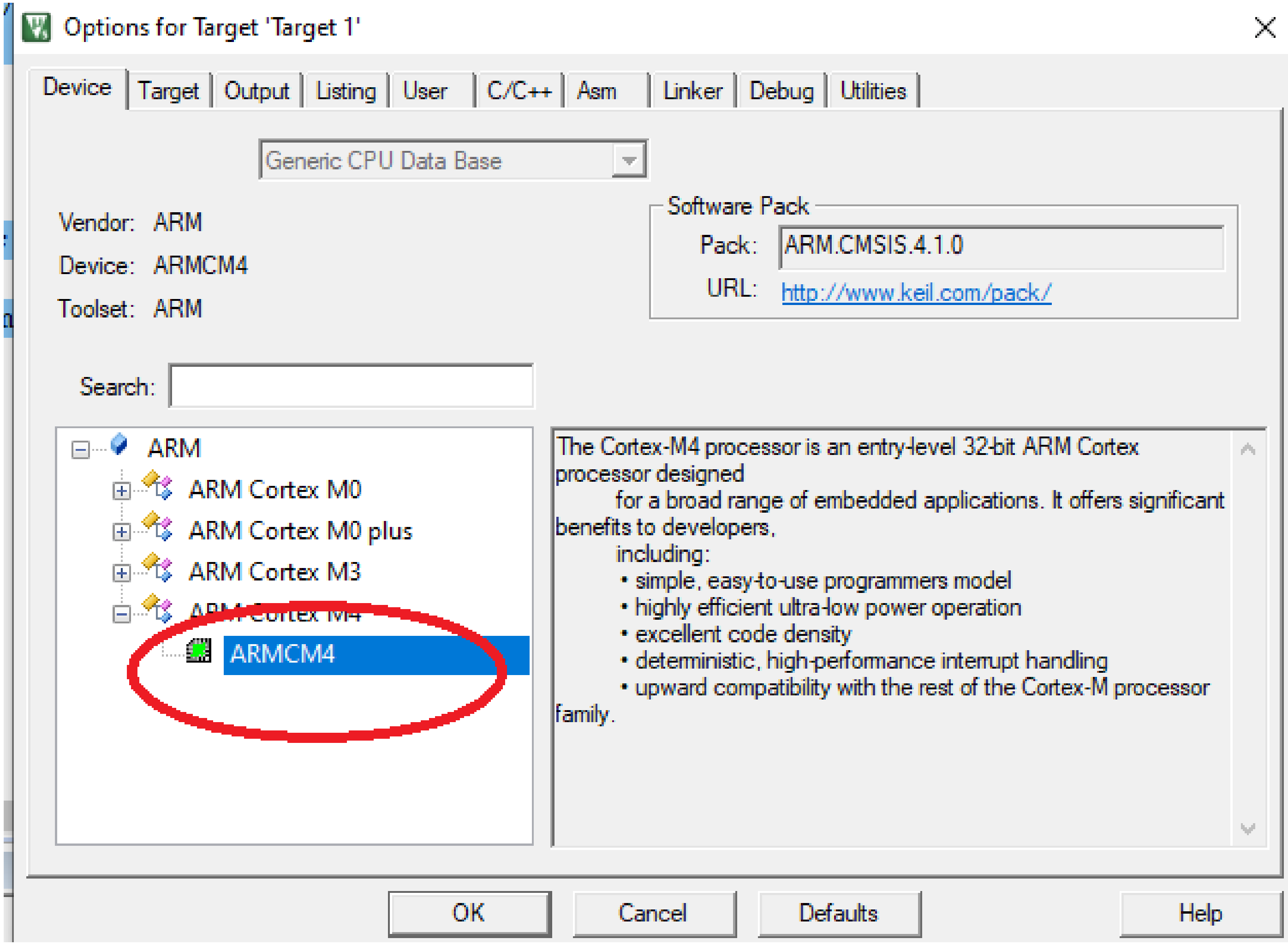
```
#define MAXSTR (100)

Typedef char LONG_STRING_t [MAXSTR+1] ;

LONG_STRING_t  line ;
LONG_STRING_t  long_string;
```

- Shorter name for struct types:

```
typedef struct  {
    LONG_STRING_t label ;     // name for the point (fixed length)
    double  x ;               // x-coordinate
    double  y ;               // y-coordinate
}  POINT_t;

POINT_t origin ;
POINT_t focus ;
POINT_t *p_point = origin;
```

Keil-C Simulator

Create Project



Vendor: ARM

Device: ARMCM4

Toolset: ARM

Software Pack

Pack: ARM.CMSIS.4.1.0

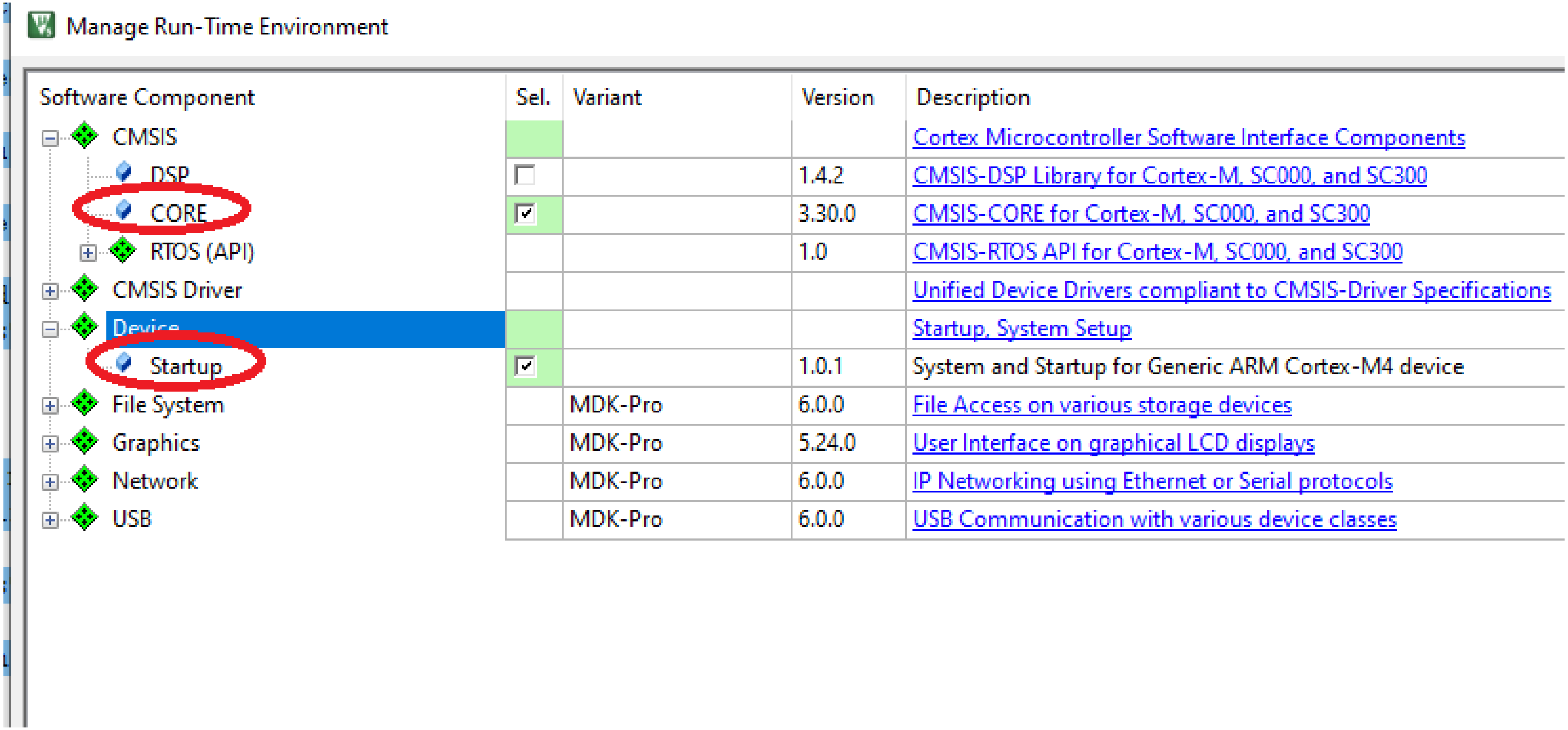
URL: http://www.keil.com/pack/

Search:

The Cortex-M4 processor is an entry-level 32-bit ARM Cortex processor designed for a broad range of embedded applications. It offers significant benefits to developers, including:

- simple, easy-to-use programmers model
- highly efficient ultra-low power operation
- excellent code density
- deterministic, high-performance interrupt handling
- upward compatibility with the rest of the Cortex-M processor family.

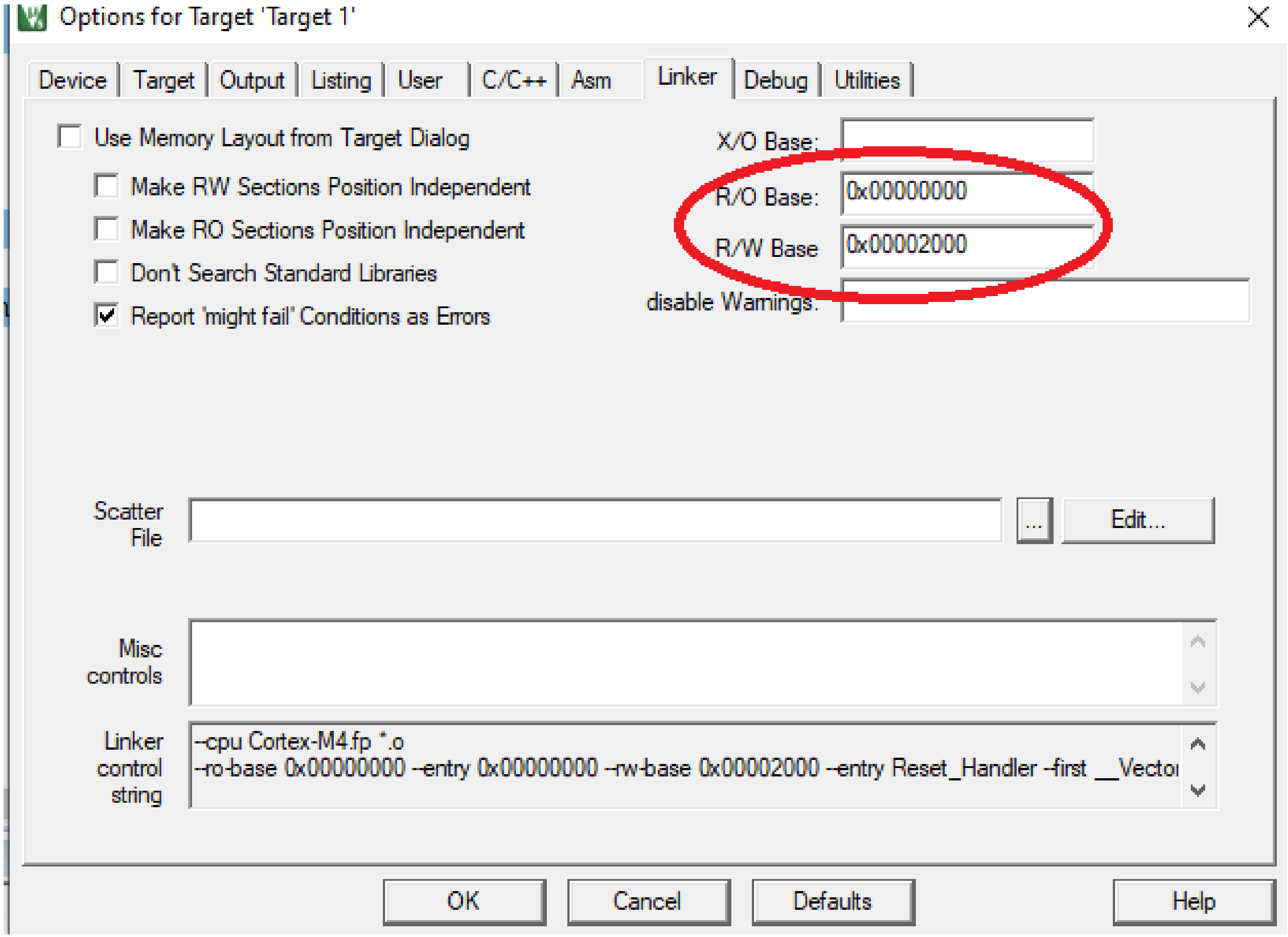
Manage Run-Time Env



Manage Run-Time Environment

Software Component	Sel.	Variant	Version	Description
CMSIS				Cortex Microcontroller Software Interface Components
DSP	□		1.4.2	CMSIS-DSP Library for Cortex-M, SC000, and SC300
CORE	☑		3.30.0	CMSIS-CORE for Cortex-M, SC000, and SC300
RTOS (API)			1.0	CMSIS-RTOS API for Cortex-M, SC000, and SC300
CMSIS Driver				Unified Device Drivers compliant to CMSIS-Driver Specifications
Device				Startup, System Setup
Startup	☑		1.0.1	System and Startup for Generic ARM Cortex-M4 device
File System		MDK-Pro	6.0.0	File Access on various storage devices
Graphics		MDK-Pro	5.24.0	User Interface on graphical LCD displays
Network		MDK-Pro	6.0.0	IP Networking using Ethernet or Serial protocols
USB		MDK-Pro	6.0.0	USB Communication with various device classes

Change Linker Option



□ Use Memory Layout from Target Dialog

□ Make RW Sections Position Independent

□ Make RO Sections Position Independent

□ Don't Search Standard Libraries

☑ Report 'might fail' Conditions as Errors

X/O Base:

R/O Base: 0x00000000

R/W Base 0x00002000

disable Warnings:

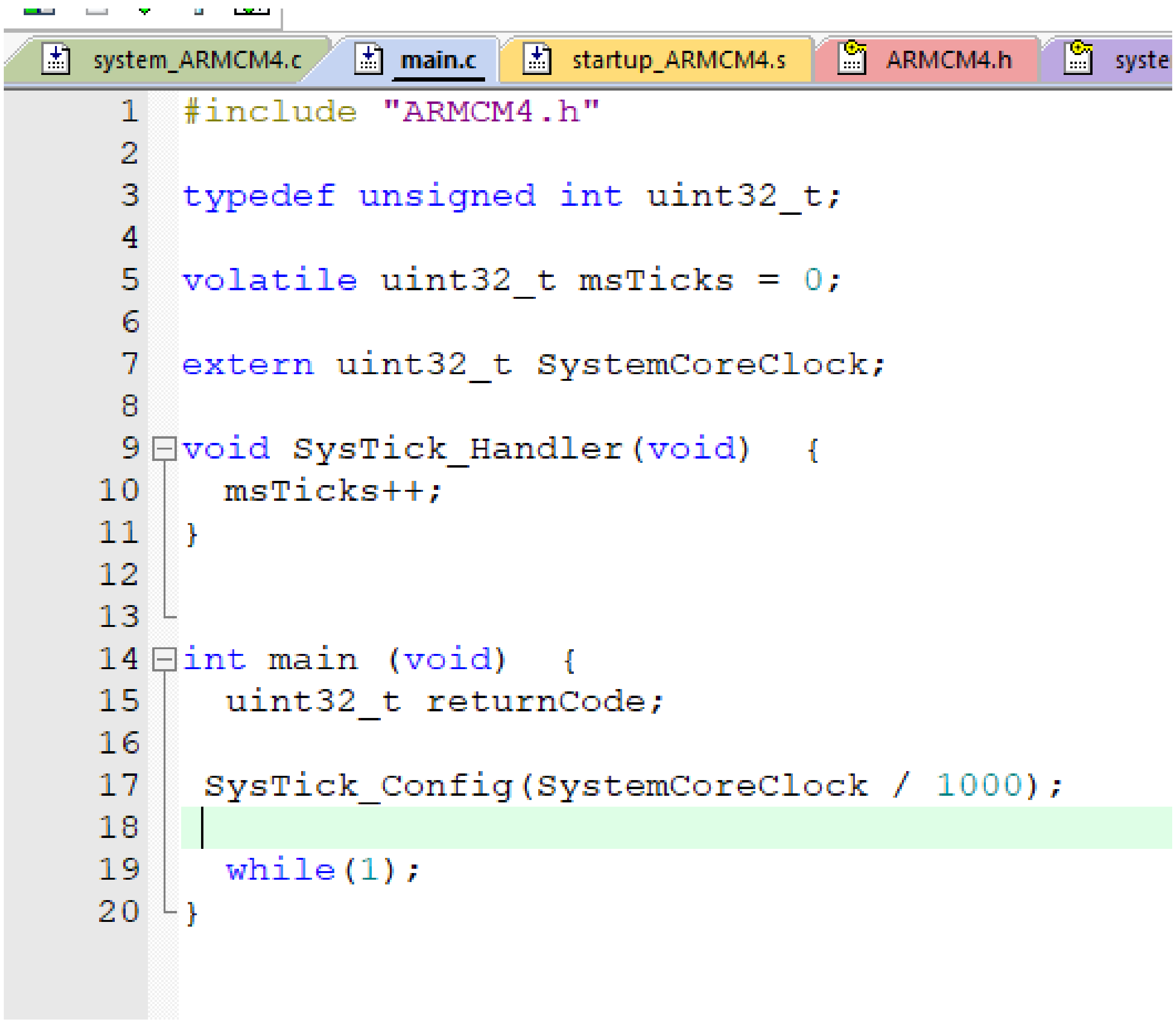
Scatter File

Misc controls

Linker control string

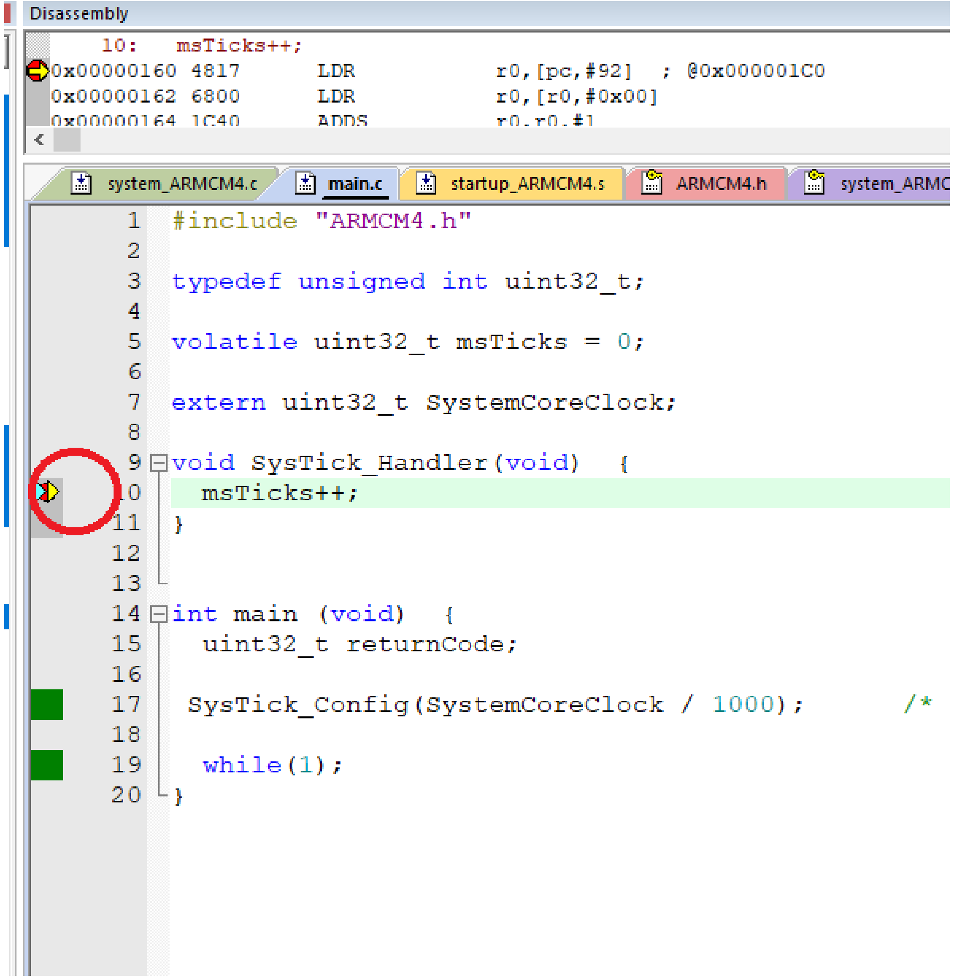
```
--cpu Cortex-M4.fp *.o
--ro-base 0x00000000 --entry 0x00000000 --rw-base 0x00002000 --entry Reset_Handler --first __Vecto
```

Example code - SysTick



```c
#include "ARMCM4.h"

typedef unsigned int uint32_t;

volatile uint32_t msTicks = 0;

extern uint32_t SystemCoreClock;

void SysTick_Handler(void)   {
  msTicks++;
}


int main (void)   {
  uint32_t returnCode;

 SysTick_Config(SystemCoreClock / 1000);

  while(1);
}
```

Keil-C – Set Breaking Point



```
    10:   msTicks++;
0x00000160 4817      LDR          r0,[pc,#92]  ; @0x000001C0
0x00000162 6800      LDR          r0,[r0,#0x00]
0x00000164 1C40      ADDS         r0,r0,#1
```

```
#include "ARMCM4.h"

typedef unsigned int uint32_t;

volatile uint32_t msTicks = 0;

extern uint32_t SystemCoreClock;

void SysTick_Handler(void)  {
  msTicks++;
}


int main (void)  {
  uint32_t returnCode;

 SysTick_Config(SystemCoreClock / 1000);       /*

  while(1);
}
```

Question Time