POLITECNICO
DI MILANO

www.polimi.it





Fundamentals of c-programming

SENSOR SYSTEMS

Dr. Federica Villa

Summary

- Fundamentals of C-programming
- Basics of Assembly
- Build process

Summary

- Fundamentals of C-programming
- Basics of Assembly
- Build process

Embedded C

C for microcontrollers:

- benefits for both low level hardware interactions and high level software language features
- portability across different embedded platforms

- efficient memory management
- timing centric operations
- direct hardware/IO control
- code size constraints
- optimized execution

→ Optimum features with minimum space and time

Declaring variables

Variable declaration format:

```
type-qualifier(s) type-modifier data-type variable-name = initial-value;
```

Examples of declaration and assignment:

```
const  unsigned  char  pippo  =  12;
long  int  pluto;
```

In the assignments:

- 0b – indicates binary
- 0x – indicates hexadecimal

e.g. `0xA49E = 0b1010010010011110`

Declaring variables

Variable declaration format:

```
type-qualifier(s) type-modifier data-type variable-name = initial-value;
```

Examples of declaration and assignment:

```
const  unsigned  char  pippo  =  12;
long  int  pluto;
```

data-types:

- integer (int)
- floating (float)
- double (double)
- char (char)
- derived (multiple values)
- user defined (structures)

Declaring variables

Variable declaration format:

```
type-qualifier(s) type-modifier data-type variable-name = initial-value;
```

Examples of declaration and assignment:

```
const  unsigned  char  pippo  =  12;
long  int  pluto;
```

types-modifier:

- long
- short
- unsigned
- signed

Declaring variables

Variable declaration format:

```
type-qualifier(s) type-modifier data-type variable-name = initial-value;
```

Examples of declaration and assignment:

```
const  unsigned  char  pippo  =  12;
long  int  pluto;
```

types-qualifier:

- const
- volatile
- restrict

C language available data

type	Bytes	min value	max value
char	1	-2^{8-1} = -128 1000 0000	$+2^{8-1}-1$ = +127 0111 1111
unsigned char	1	0 0000 0000	$2^{8}-1$ = 255 1111 1111
short	2	-2^{16-1} = -32,768 1000 0000 0000 0000	$+2^{16-1}-1$ = +32,767 0111 1111 1111 1111
unsigned short	2	0 0000 0000 0000 0000	$2^{16-1}-1$ = 65,535 1111 1111 1111 1111
int	4	-2^{32-1} = -2,147,483,648 0x8000 0000	$+2^{32-1}-1$ = +2,147,483,647 0x7FFF FFFF
unsigned int	4	0 0x0000 0000	4,294,967,295 0xFFFF FFFF
long	4	-2^{32-1} = -2,147,483,648 0x8000 0000	$+2^{32-1}-1$ = +2,147,483,647 0x7FFF FFFF
unsigned long	4	0 0x0000 0000	$2^{32}-1$ = 4,294,967,295 0xFFFF FFFF
long long	8	-2^{64-1} = -9,223,372,036,854,775,808 0x8000 0000 0000 0000	$-2^{64-1}-1$ = 9,223,372,036,854,775,807 0x7FFF FFFF FFFF FFFFG
unsigned long long	8	0 0x0000 0000 0000 0000	$2^{64}-1$ = 18,446,744,073,709,551,615 0xFFFF FFFF FFFF FFFF
float	4	-3.4E+38 6 digit precision	+3.4E+38 6 digit precision
double	8	-1.7E+308 15 digit precision	+1.7E+308 15 digit precision
long double	10	-1.1E+4932 19 digit precision	+1.1E+4932 19 digit precision

Examples

A constant floating point number must contain "." (e.g. `3.0`):

```
19/10          resulting in 1              being integer
19.0/10.0      resulting in 1.9            being floating point
```

```
myfloat = 1.0 / 2.0;          /* assigns a floating 0.5     */
myint = 1 / 3;                /* assigns an integer 0       */
myfloat = (1 / 2) + (1 / 2);  /* assign floating 0.0        */
myfloat = 3.0 / 2.0;          /* assign floating 1.5        */
myint = myfloat;              /* assign integer 1           */
```

```
short int zipcode;     /* zipcode is a short integer                  */
zipcode = 50121;       /* let's go to Florence for a while  */
                       /* instead no, due to truncation and sign bit…*/
                       /* … it results in -15415            */
```

Type conversions and *cast*

`Along=Bint*Clong;` `Bint` is converted to long before *

`Cdouble=Bint/Ddouble;` `Bint` is converted to double /

Automatic conversion order: `float > long > int > short > char`

To force conversion, use cast operator enclosed between ()

`Along=(int)Clong;` converts `Clong` to int and assigns it to `Along`

`myintvar=(int)3.14159;` assigns the value 3 to the int

```
int  intvar = 4567;
char character;
character = (char)intvar;
```

put the LSbyte of int into the char, discarding the upper 24 bits

Operators

Used to assign, manipulate and compare data

Logical operators:	&&	||	^^	~		
	and	or	xor	not		
Bitwise (bit-by-bit) operators:	&	|	^	~	>>	<<
	and	or	xor	one's complement	shift right	shift left
Arithmetic:	+	-	*	/	%	
Relational:	<	<=	>	>=	==	!=

Increment and Decrement operators

C is concise yet unambiguous thanks to powerful short-hand operators:

`++` `--`

`pulses = pulses + 1;` use the more compact notation `++pulses;`

Both operators can either precede or follow the variable.

`++pulses` the increment will be performed **before** any use

`final = 3 * ++pulses;` final=12 pulses=4 (if pulse was 3)

`pulses++` the increment will be performed **after** any use

`final = 3 * pulses--;` final=9 pulses=2 (if pulse was 3)

Operators and assignment

Operators can also be used in combination with assignments.

The operators that can be used in assignment operators are

```
+    -    *    /    %    <<    >>    &    ^    |
```

Given two expressions and a generic binary operator Q:

`exprA Q= exprB;` is equivalent to `exprA = (exprA) Q (exprB);`

Examples:

```
B += A;   →   B = B + A ;
B |= A;   →   B = B | A ;
```

Example: masking bits

To Set, Clear, Complement individual bits:

```
C = A & 0xFE;      A      a b c d e f g h
                   0xFE   1 1 1 1 1 1 1 0
                   C      a b c d e f g 0   Clear selected bit of A

C = A & 0x01;      A      a b c d e f g h
                   0x01   0 0 0 0 0 0 0 1
                   C      0 0 0 0 0 0 0 h   Clear all but the selected bit of A

C = A | 0x01;      A      a b c d e f g h
                   0x01   0 0 0 0 0 0 0 1
                   C      a b c d e f g 1   Set selected bit of A

C = A ^ 0x01;      A      a b c d e f g h
                   0x01   0 0 0 0 0 0 0 1
                   C      a b c d e f g h'  Complement selected bit of A
```

To Shift, Multiply and Divide nibbles and bytes:

```
B = '1';                          B =   0 0 1 1 0 0 0 1   (ASCII 0x31)
C = '3';                          C =   0 0 1 1 0 0 1 1   (ASCII 0x35)
D = (B << 4) | (C & 0x0F);
                        (B << 4) =   0 0 0 1 0 0 0 0
                      (C & 0x0F) =   0 0 0 0 0 0 1 1
                               D =   0 0 0 1 0 0 1 1   (Packed BCD 0x13)
```

Pointers

The pointer type in C is the common indirect addressing in Assembly

Pointer declaration:

```
int *ptr;
```

The unary ***dereference*** operator * gives the content of the cell pointed to by a pointer; the operator & gives the address of a variable.

Pointer assignment to the variable's address: `ptr = &content;`

Value assignment from a pointed cell: `content = *ptr;`

Control program flow

Conditional expression

```
if (conditionA) {
   //code A
}

else if (condition){
   //code B
}

else {
   //code C
}
```

Switch - case

```
switch (cexpression) {
   case const-expA:
        //code A
        break;
   case const-expB:
        //code B
        break;
   …
   default:
        //code C
        break;
}
```

Loops

For loop

```
for ( variable-initialize; condition; variable-expression ) {
   //code
}
```

preloop check

While loop

```
while ( condition ) {
   //code
}
```

preloop check

Do… while loop

```
do {
   //code
} while ( condition )
```

postloop check

Break and continue

break → exit loop immediately

continue → moves to next iteration

Example:

```
for( i = 0; i < 100; i++ ) {
   if ( array[i] < 0) {
        continue;
   }
   if ( array[i] == 0 {
        break;
   }
   sum += array[i];
}
//code
```

Functions

Function declaration:

```
function-type function-name (param1-type param1, … )
```

Example:

```
int sum (int a, int b);
```

Function definition:

```
function-type function-name (param1-type param1, … ) {
   //code for the function
}
```

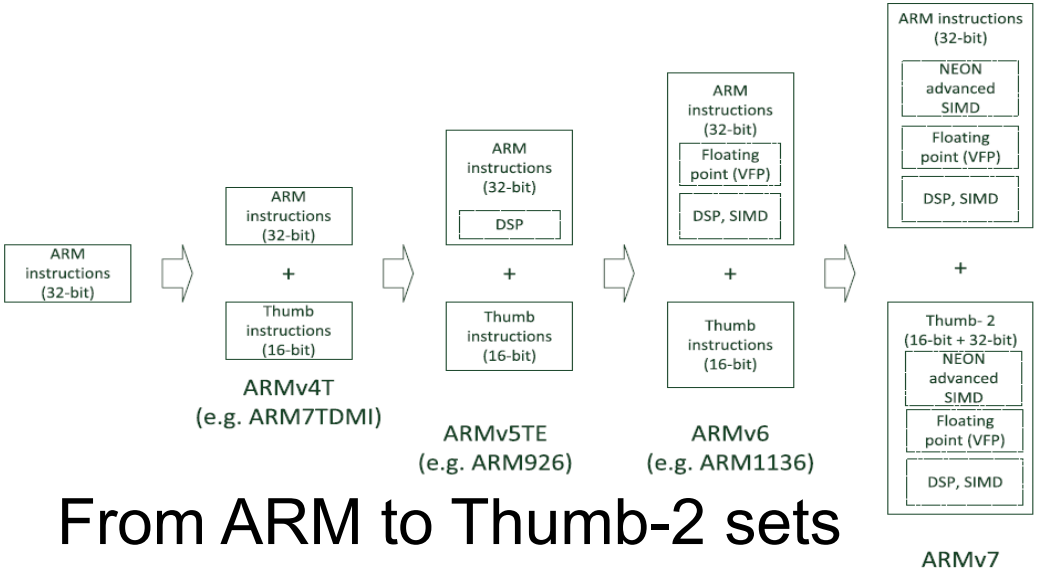
Example:

```
int sum (int a, int b) {
   return ( a + b);
}
```

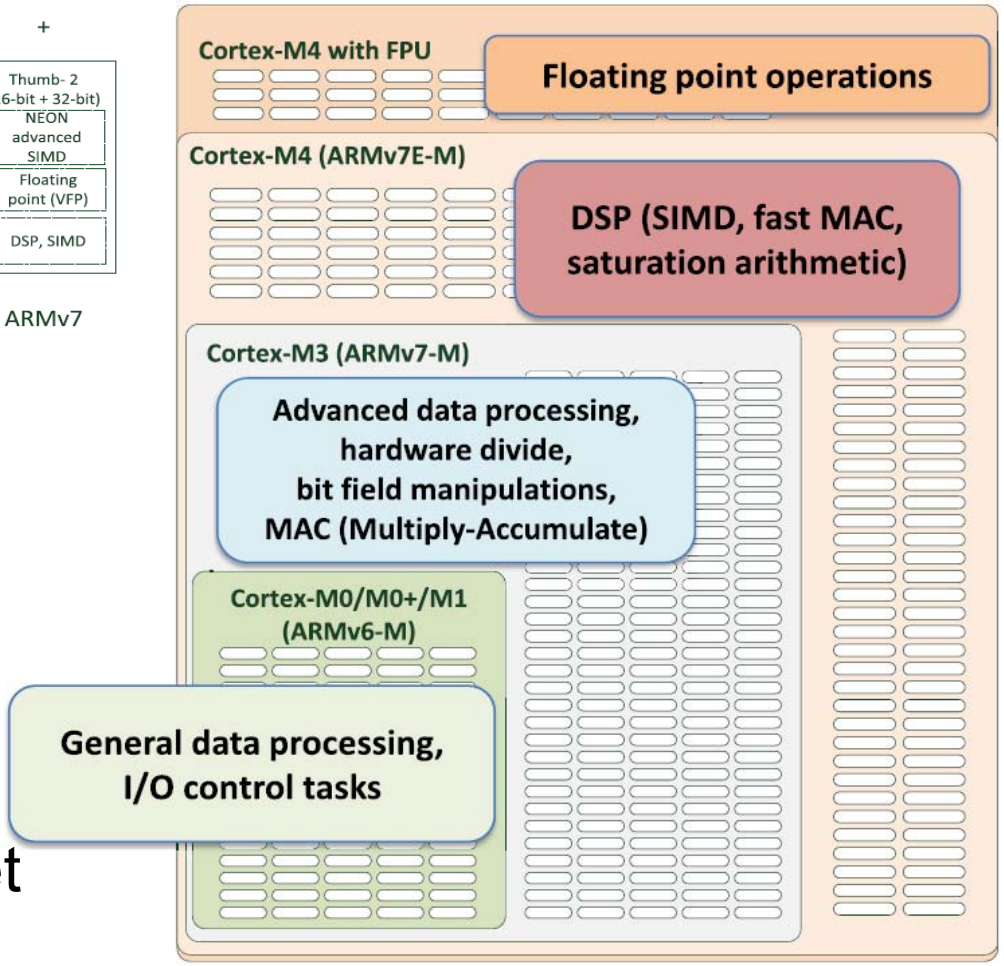
Summary

- Fundamentals of C-programming
- Basics of Assembly
- Build process

ARM-Cortex M instruction set

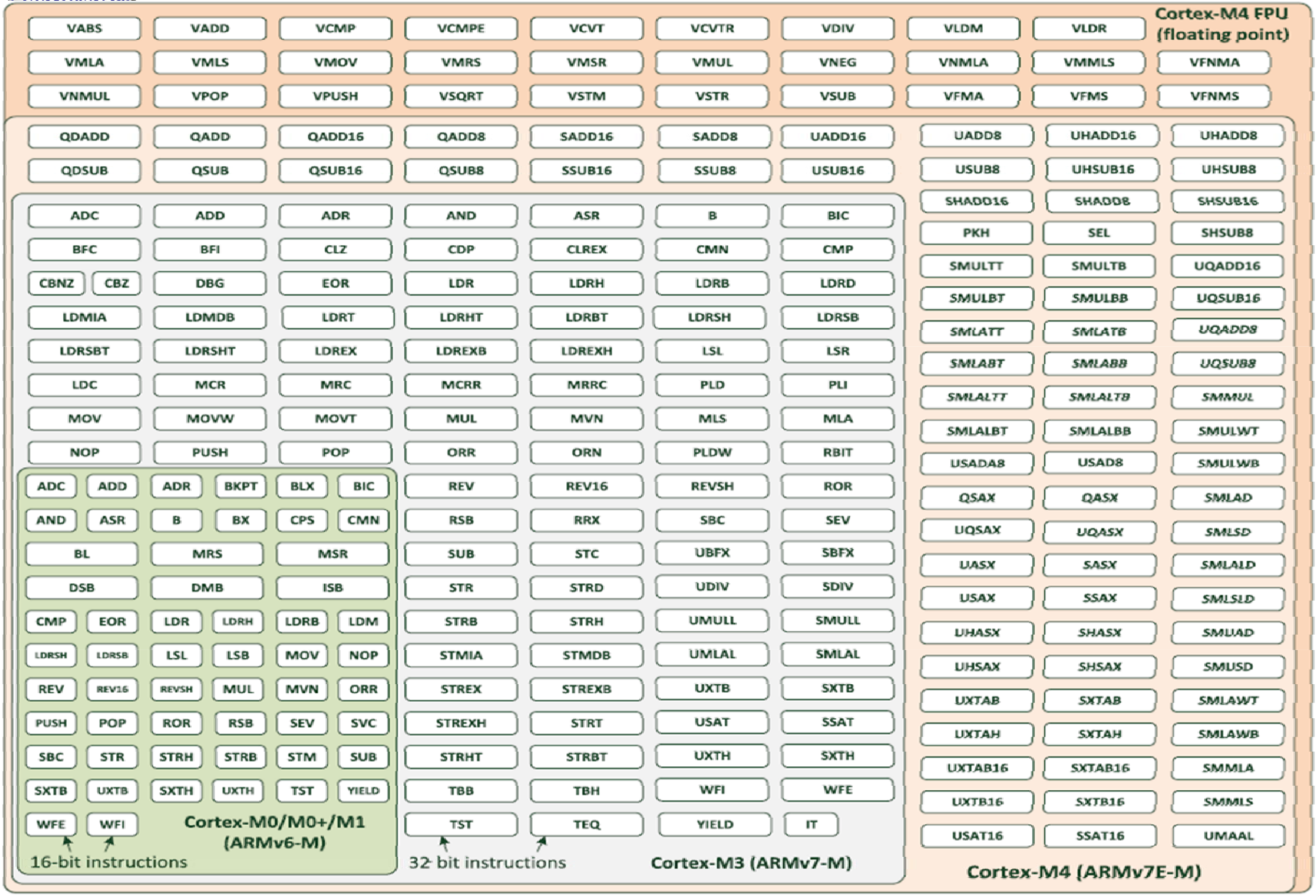


From ARM to Thumb-2 sets



Cortex M Thumb-2 set

ARM-Cortex M instruction set



ARM-Cortex Assembly language

Assembly format

```
label
        mnemonic operand1, operand2, …          ; Comments
```

Immediate data are usually prefixed with "#"

```
MOVS R0, #0x12              ; Set R0 = 0x12 (hexadecimal)
MOVS R1, #'A'               ; Set R1 = ASCII character A
```

Instruction-width selection

```
BCS.W   label    ;creates a 32bit instr. even for a short branch

ADDS.W R0, R0, R1 ;creates a 32bit instr. even though the same
                  ;operation can be done by a 16bit instr.
```

ARM-Cortex M3 instruction set 1/6

Mnemonic	Operands	Brief description	Flags
ADC, ADCS	{Rd,} Rn, Op2	Add with Carry	N,Z,C,V
ADD, ADDS	{Rd,} Rn, Op2	Add	N,Z,C,V
ADD, ADDW	{Rd,} Rn, #imm12	Add	N,Z,C,V
ADR	Rd, label	Load PC-relative Address	-
AND, ANDS	{Rd,} Rn, Op2	Logical AND	N,Z,C
ASR, ASRS	Rd, Rm, <Rs|#n>	Arithmetic Shift Right	N,Z,C
B	label	Branch	-
BFC	Rd, #lsb, #width	Bit Field Clear	-
BFI	Rd, Rn, #lsb, #width	Bit Field Insert	-
BIC, BICS	{Rd,} Rn, Op2	Bit Clear	N,Z,C
BKPT	#imm	Breakpoint	-
BL	label	Branch with Link	-
BLX	Rm	Branch indirect with Link	-
BX	Rm	Branch indirect	-
CBNZ	Rn, label	Compare and Branch if Non Zero	-
CBZ	Rn, label	Compare and Branch if Zero	-
CLREX	-	Clear Exclusive	-
CLZ	Rd, Rm	Count Leading Zeros	-
CMN	Rn, Op2	Compare Negative	N,Z,C,V
CMP	Rn, Op2	Compare	N,Z,C,V

ARM-Cortex M3 instruction set 2/6

Mnemonic	Operands	Brief description	Flags
ISB	-	Instruction Synchronization Barrier	-
IT	-	If-Then condition block	-
LDM	Rn{!}, reglist	Load Multiple regs, increment after	-
LDMDB, LDMEA	Rn{!}, reglist	Load Multiple regs, decrement before	-
LDMFD, LDMIA	Rn{!}, reglist	Load Multiple regs, increment after	-
LDR	Rt, [Rn, #offset]	Load Register with word	-
LDRB, LDRBT	Rt, [Rn, #offset]	Load Register with byte	-
LDRD	Rt, Rt2, [Rn, #offset]	Load Register with two bytes	-
LDREX	Rt, [Rn, #offset]	Load Register Exclusive	-
LDREXB	Rt, [Rn]	Load Register Exclusive with Byte	-

ARM-Cortex M3 instruction set 3/6

Mnemonic	Operands	Brief description	Flags
LDREXH	Rt, [Rn]	Load Register Exclusive with Half-word	-
LDRH,LDRHT	Rt, [Rn, #offset]	Load Register with Half-word	-
LDRSB,LDRSBT	Rt, [Rn, #offset]	Load Register with Signed Byte	-
LDRSH,LDRSHT	Rt, [Rn, #offset]	Load Register with Signed Half-word	-
LDRT	Rt, [Rn, #offset]	Load Register with word	-
LSL, LSLS	Rd, Rm, <Rs|#n>	Logical Shift Left	N,Z,C
LSR, LSRS	Rd, Rm, <Rs|#n>	Logical Shift Right	N,Z,C
MLA	Rd, Rn, Rm, Ra	Multiply & Accumulate, 32-bit result	-
MLS	Rd, Rn, Rm, Ra	Multiply and Subtract, 32-bit result	-
MOV, MOVS	Rd, Op2	Move	N,Z,C
MOVT	Rd, #imm16	Move Top	-
MOVW, MOV	Rd, #imm16	Move 16-bit constant	N,Z,C
MRS	Rd, spec_reg	Move from Special Reg to general reg	-
MSR	spec_reg, Rm	Move from general reg to Special Reg	N,Z,C,V
MUL, MULS	{Rd,} Rn, Rm	Multiply, 32-bit result	N,Z

ARM-Cortex M3 instruction set 4/6

Mnemonic	Operands	Brief description	Flags
MVN, MVNS	Rd, Op2	Move NOT	N,Z,C
NOP	-	No Operation	-
ORN, ORNS	{Rd,} Rn, Op2	Logical OR NOT	N,Z,C
ORR, ORRS	{Rd,} Rn, Op2	Logical OR	N,Z,C
POP	reglist	Pop registers from stack	-
PUSH	reglist	Push registers onto stack	-
RBIT	Rd, Rn	Reverse Bits	-
REV	Rd, Rn	Reverse byte order in a word	-
REV16	Rd, Rn	Reverse byte order in each half-word	-
REVSH	Rd, Rn	Reverse byte order in bottom half-word and sign extend	-
ROR, RORS	Rd, Rm, <Rs|#n>	Rotate Right	N,Z,C
RRX, RRXS	Rd, Rm	Rotate Right with Extend	N,Z,C
RSB, RSBS	{Rd,} Rn, Op2	Reverse Subtract	N,Z,C,V
SBC, SBCS	{Rd,} Rn, Op2	Subtract with Carry	N,Z,C,V
SBFX	Rd, Rn, #lsb, #width	Signed Bit Field Extract	-
SDIV	{Rd,} Rn, Rm	Signed Divide	-
SEV	-	Send Event	-

ARM-Cortex M3 instruction set 5/6

Mnemonic	Operands	Brief description	Flags
		Accumulate (32x32 + 64), 64-bit result	
SMULL	RdLo, RdHi, Rn, Rm	Signed Multiply (32x32),64-bit result	-
SSAT	Rd, #n, Rm {,shift #s}	Signed Saturate	Q
STM	Rn{!}, reglist	Store Multiple regs, increment after	-
STMDB, STMEA	Rn{!}, reglist	Store Multiple regs, decrement before	-
STMFD, STMIA	Rn{!}, reglist	Store Multiple regs, increment after	-
STR	Rt, [Rn, #offset]	Store Register word	-
STRB, STRBT	Rt, [Rn, #offset]	Store Register byte	-
STRD	Rt, Rt2, [Rn, #offset]	Store Register two words	-
STREX	Rd, Rt, [Rn, #offset]	Store Register Exclusive	-
STREXB	Rd, Rt, [Rn]	Store Register Exclusive Byte	-
STREXH	Rd, Rt, [Rn]	Store Register Exclusive Half-word	-
STRH, STRHT	Rt, [Rn, #offset]	Store Register Half-word	-
STRT	Rt, [Rn, #offset]	Store Register word	-

ARM-Cortex M3 instruction set 6/6

Mnemonic	Operands	Brief description	Flags
SUB, SUBS	{Rd,} Rn, Op2	Subtract	N,Z,C,V
SUB, SUBW	{Rd,} Rn, #imm12	Subtract	N,Z,C,V
SVC	#imm	Supervisor Call	-
SXTB	{Rd,} Rm {,ROR #n}	Sign extend a byte	-
SXTH	{Rd,} Rm {,ROR #n}	Sign extend a half-word	-
TBB	[Rn, Rm]	Table Branch Byte	-
TBH	[Rn, Rm, LSL #1]	Table Branch Half-word	-
TEQ	Rn, Op2	Test Equivalence	N,Z,C
TST	Rn, Op2	Test	N,Z,C
UBFX	Rd, Rn, #lsb, #width	Unsigned Bit Field Extract	-
UDIV	{Rd,} Rn, Rm	Unsigned Divide	-
UMLAL	RdLo, RdHi, Rn, Rm	Unsigned Multiply with Accumulate (32x32 + 64), 64-bit result	-
UMULL	RdLo, RdHi, Rn, Rm	Unsigned Multiply (32x32),64bit res.	-
USAT	Rd, #n, Rm {,shift #s}	Unsigned Saturate	Q
UXTB	{Rd,} Rm {,ROR #n}	Zero extend a Byte	-
UXTH	{Rd,} Rm {,ROR #n}	Zero extend a Half-word	-
WFE	-	Wait For Event	-
WFI	-	Wait For Interrupt	-

ARM-Cortex instruction options

Conditional execution

Based on the APSR (Application Program Status Register) flags:

- **N** "negative" 1 when the result of the operation was negative
- **Z** "zero" 1 when the result of the operation was zero
- **C** "carry" 1 when the operation resulted in a carry
  if an addition is greater than or equal to 2^{32}
  if a subtraction is positive or zero
  as the result of an inline barrel shifter
- **V** "overflow" 1 when the operation caused overflow
  when the sign of the result (bit[31]) does not match
  (e.g. if adding two negative values results in positive)

```
0x70000000 + 0x70000000      = 0xE0000000,      N=1, Z=0, C=0, V=1
0x90000000 + 0x90000000      = 0x30000000,      N=0, Z=0, C=1, V=1
0x80000000 + 0x80000000      = 0x00000000,      N=0, Z=1, C=1, V=1
0x00001234 - 0x00001000      = 0x00000234,      N=0, Z=0, C=1, V=0
0x00000004 - 0x00000005      = 0xFFFFFFFF,      N=1, Z=0, C=0, V=0
0xFFFFFFFF - 0xFFFFFFFC      = 0x00000003,      N=0, Z=0, C=1, V=0
0x80000005 - 0x80000004      = 0x00000001,      N=0, Z=0, C=1, V=0
0x70000000 - 0xF0000000      = 0x80000000,      N=1, Z=0, C=0, V=1
0xA0000000 - 0xA0000000      = 0x00000000,      N=0, Z=1, C=1, V=0
```

ARM-Cortex instruction options

Conditional codes

cond	code	Meaning (int)	Meaning (FP)	Flags
0000	EQ	Equal	Equal	Z==1
0001	NE	Not equal	Not equal or unordered	Z==0
0010	CS	Carry set	Greater than, equal, or unordered	C==1
0011	CC	Carry clear	Less than	C==0
0100	MI	Minus, negative	Less than	N==1
0101	PL	Plus, positive or 0	Greater than, equal, or unordered	N==0
0110	VS	Overflow	Unordered	V==1
0111	VC	No overflow	Not unordered	V==0
1000	HI	Unsigned higher	Greater than, or unordered	C==1 and Z==0
1001	LS	Unsigned lower or same	Less than or equal	C==0 or Z==1
1010	GE	Signed greater than or equal	Greater than or equal	N==V
1011	LT	Signed less than	Less than, or unordered	N!=V
1100	GT	Signed greater than	Greater than	Z==0 and N==V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z==1 or N!=V
1110	(AL)	Always (uncond.)	Always (uncond.)	Any

For example, find the absolute value of a number, like R0 = abs(R1):

```
MOVS    R0, R1          ; R0 = R1 then update flags
IT      MI              ; skip next instruction if 0 or positive
RSBMI   R0, R0, #0      ; if negative then R0 = -R0
```

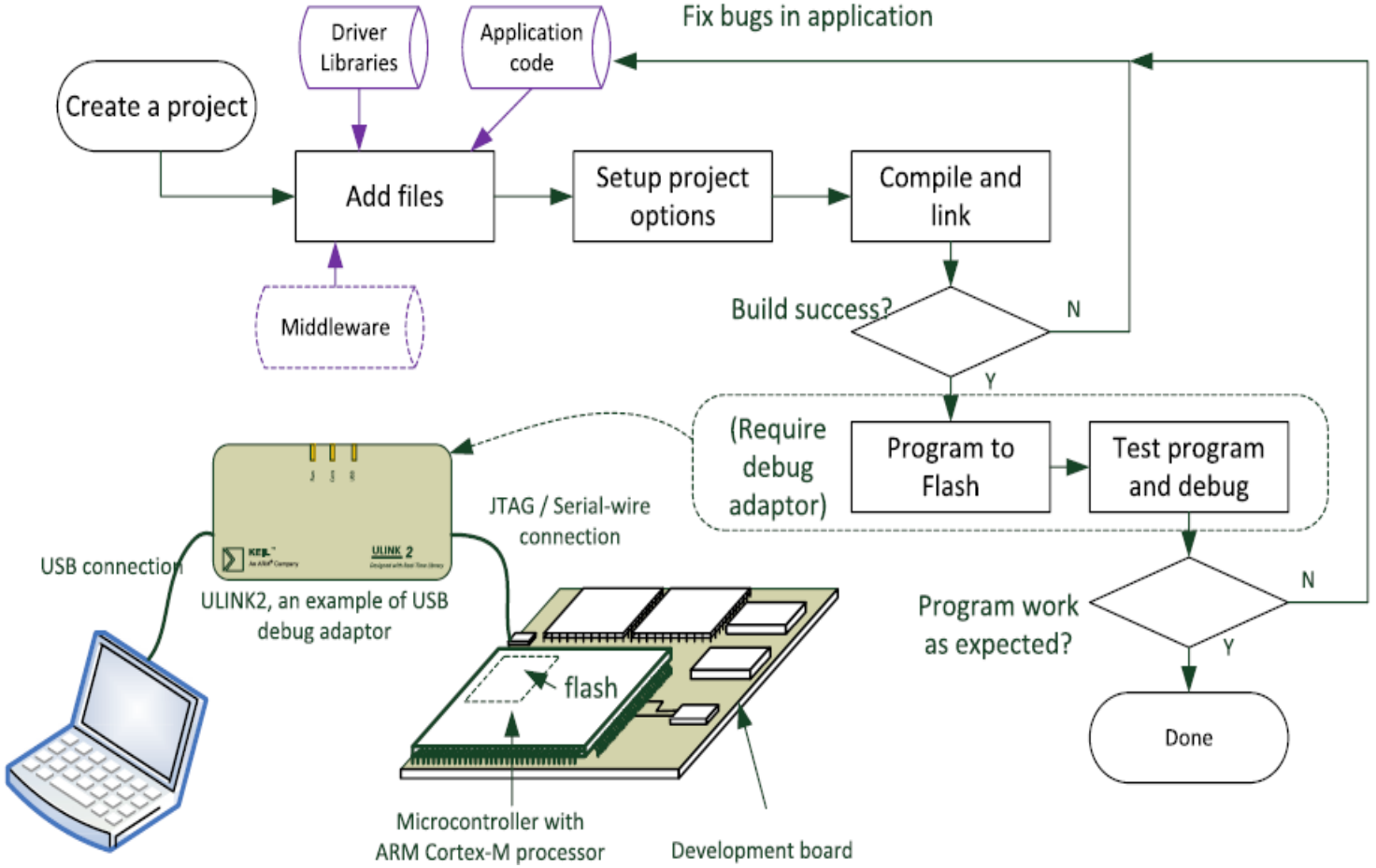
For example update the value of R4 if the signed values R0 is greater than R1 and R2 is greater than R3:

```
CMP     R0, R1          ; compare R0 and R1 then sets flags
ITT     GT              ; skip next two instructions unless GT condition holds
CMPGT   R2, R3          ; if GT ('greater than'), compares R2 and R3 and sets flags
MOVGT   R4, R5          ; if still GT then R4 = R5
```

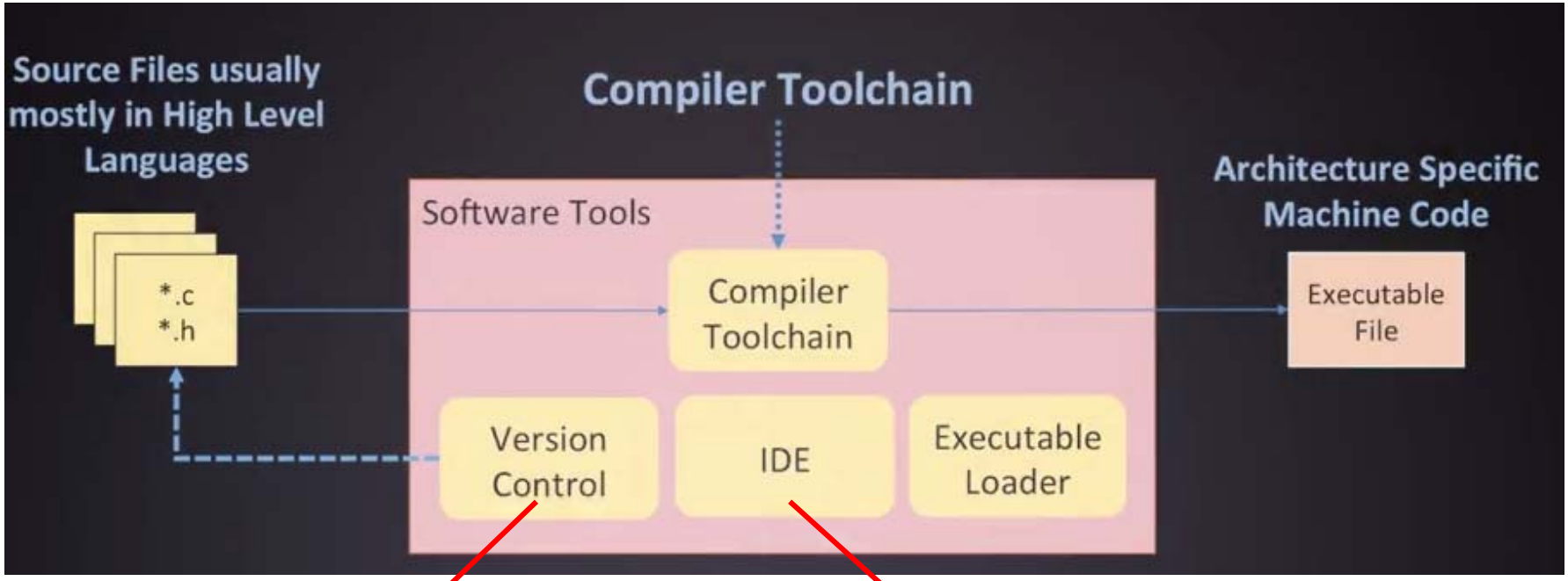
Summary

- Fundamentals of C-programming
- Basics of Assembly
- Build process

Firmware development flow



Software tools



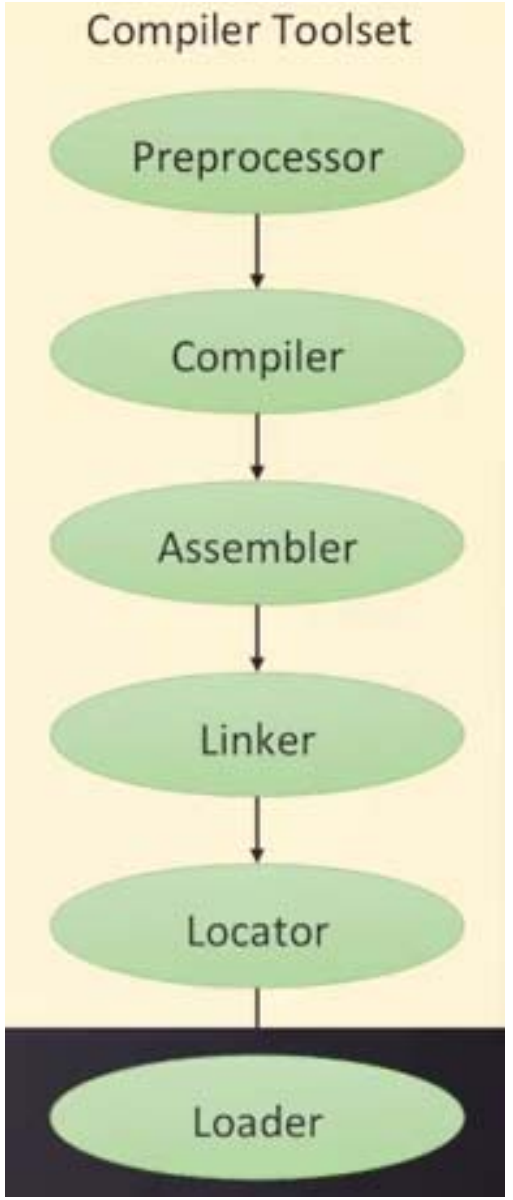
e.g.: GIT

Integrated Development Environment
(often based on Eclipse)
OR
Command Line (Linux: GCC)

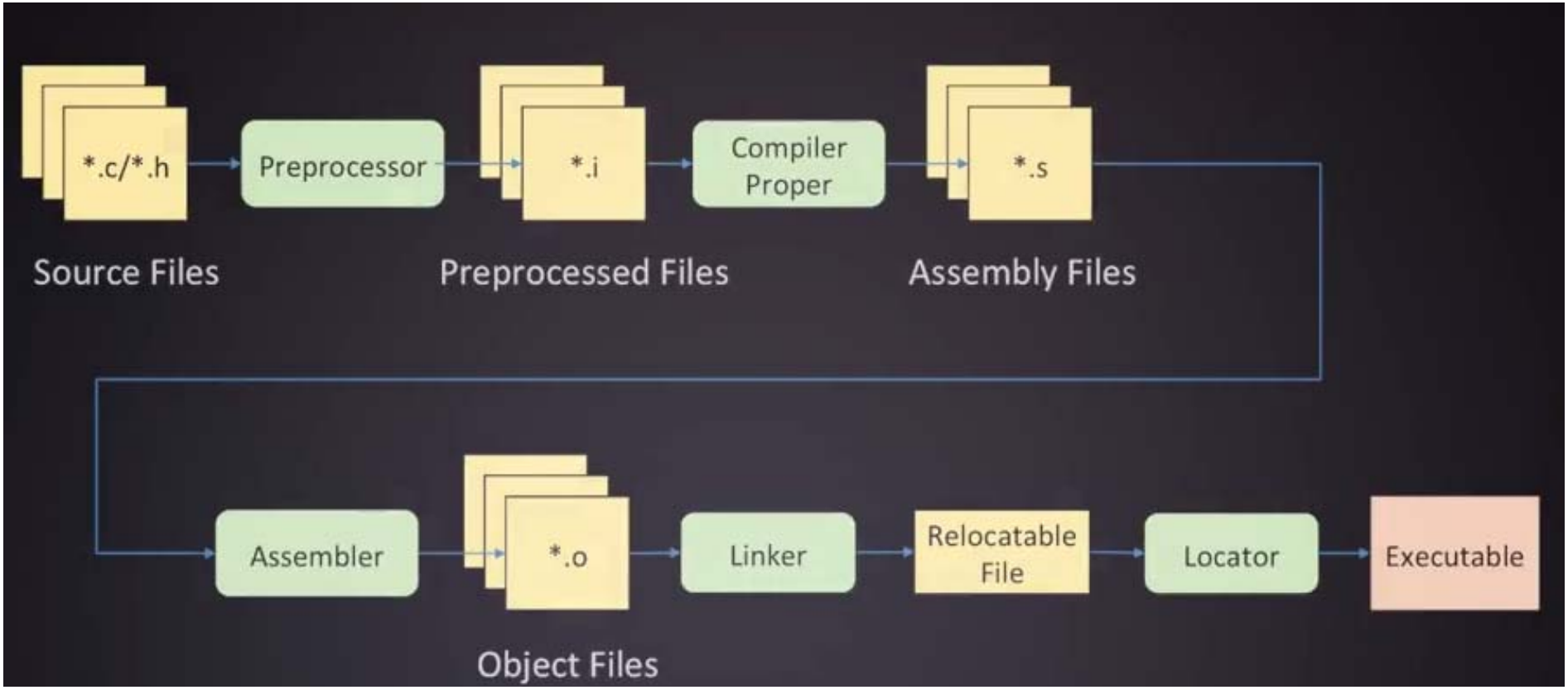
Toolchain

Build and install process:

- preprocessing
- assembling
- compiling
- linking
- locating
- installing



Build process



Translate high level languages

into architecture specific languages

Source files

- Implementation files (*.c): contain the function definitions or the actual implementation details

memory.c

```
#include “memory.h”
char mem(char * ptr, int length){
     int i;
     for ( i = 0; i < length; i++ ) {
          *ptr++ = 0;
     }
}
```

- Header files (*.h): contain the function declarations, macros, derived data type definitions (structs, enums)

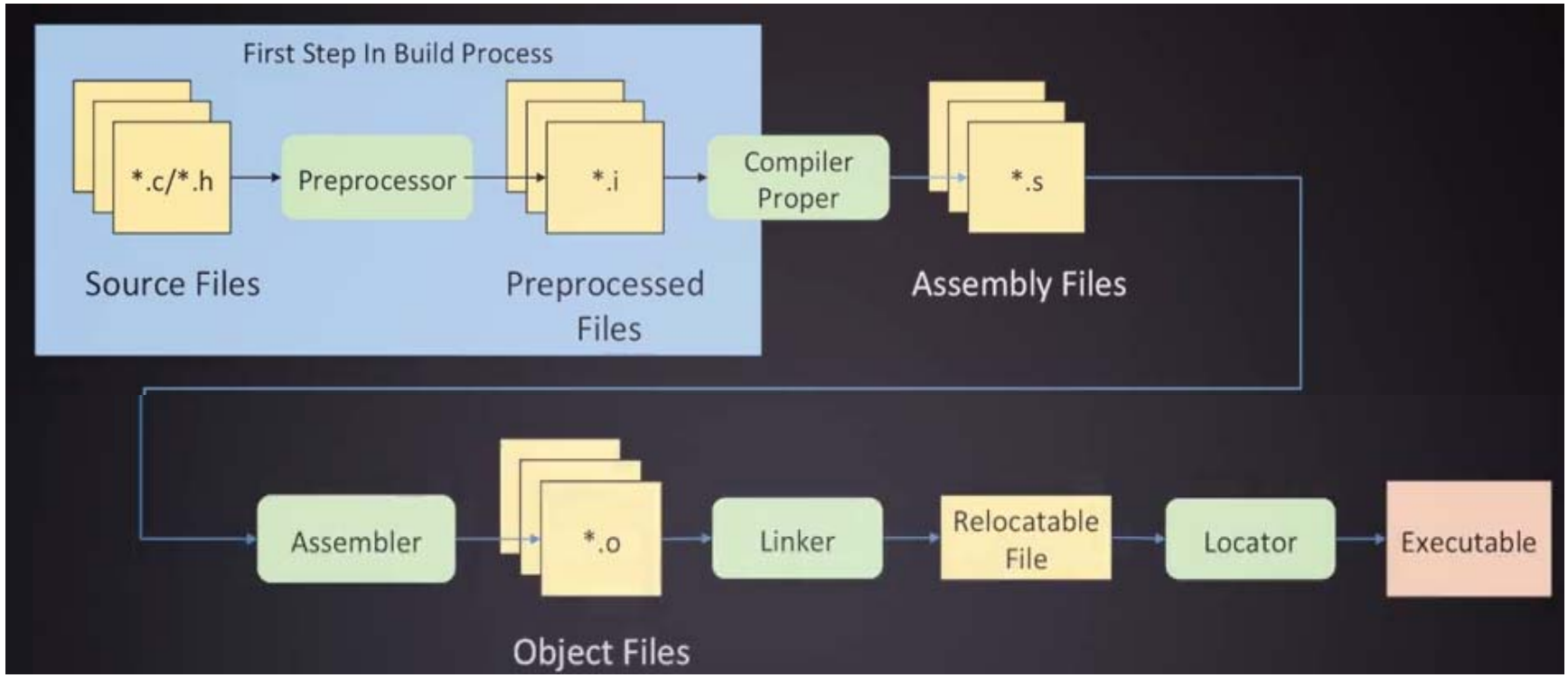
memory.h

```
#ifndef MEMORY_H

#define MEMORY_H
char mem(char * ptr, int length);

#endif
```

The preprocessor



From .c and .h generates .i files:

- evaluate preprocessor directives
- perform macro substitution

Preprocessor Directives

- special keywords used by the preprocessor before compilation
- directives start with ‘#’ sign
- Important directives:
  - `#define, #undef`
  - `#ifndef, #ifdef, #endif`
  - `#include`
  - `#warning, #error`
  - `#pragma`

#define directive

- used for defining constants:

```
#define NAME (VALUE)
```

e.g.: `#define LENGTH (10)`

- used for defining macro functions:

```
#define NAME(PARAMS) (OPERATION)
```

e.g.: `#define SQUARE(x) (x*x)`

```
#define LENGTH (10)
#define SQUARE(x) (x*x)

main(){
     char arr[LENGTH];
     int y_sqrd;
     int y;
     …
     y_sqrd = SQUARE(y);
     …
}
```

preprocessor →

```
main(){
     char arr[10];
     int y_sqrd;
     int y;
     …
     y_sqrd = (y*y);
     …
}
```

#include directive

file.c

```
#include “my_file.h”
char arr[LENGTH]

void clear(char * ptr, int size){
    int i;
    for (i = 0, i < size, i++){
        ptr[i] = 0;
    }
}
```

file.h

```
#define LENGTH (10)

void clear(char * ptr, int size);
```

preprocessor

file.i

```
void clear(char * ptr, int size);
char arr[10]

void clear(char * ptr, int size){
    int i;
    for (i = 0, i < size, i++){
        ptr[i] = 0;
    }
}
```

#if-else directives

- Conditionally compile block of code (“turn off” large amounts of code):
  - #ifded
  - #ifndef
  - #elif
  - #else
  - #endif

```
#define COMPILE_CODE

#ifdef COMPILE_CODE
       //Code will be compiled
#endif
```

```
#undef DO_NOT_COMPILE_CODE

#ifdef DO_NOT_COMPILE_CODE
       // Code will NOT be compiled
#endif
```

Example:

```
int main(void){
   #ifdef ( STM32_PLATFORM ) && ( ! TEXAS_PLATFORM )
      stm32_initialize();
   #elif ( ! STM32_PLATFORM ) && (TEXAS_PLATFORM )
      texas_initialize();
   #else
      #error “Please specify one platform target”
   #endif
   //more code here
}
```

Include guards

Top of Header file contains a `#ifndef` statement to protect against redundant includes.

Example:

memory.c

```
#include “memory.h”
#include “memory.h”

char mem(char * ptr, int length){
     int i;
     for(i = 0;i<length; i++){
          *ptr++ = 0;
     }
}
```

memory.h

```
#ifndef MEMORY_H

#define MEMORY_H
char mem(char * ptr, int length);

#endif
```

memory.i

```
char mem(char * ptr, int length);

char mem(char * ptr, int length){
     int i;
     for(i = 0;i<length; i++){
          *ptr++ = 0;
     }
}
```

#pragma directive

- Gives a specific instruction to the compiler from code itself (controls compilation from software instead of command line)
- Implementation/Compiler specific
  → unrecognized pragmas will be ignored

Examples:

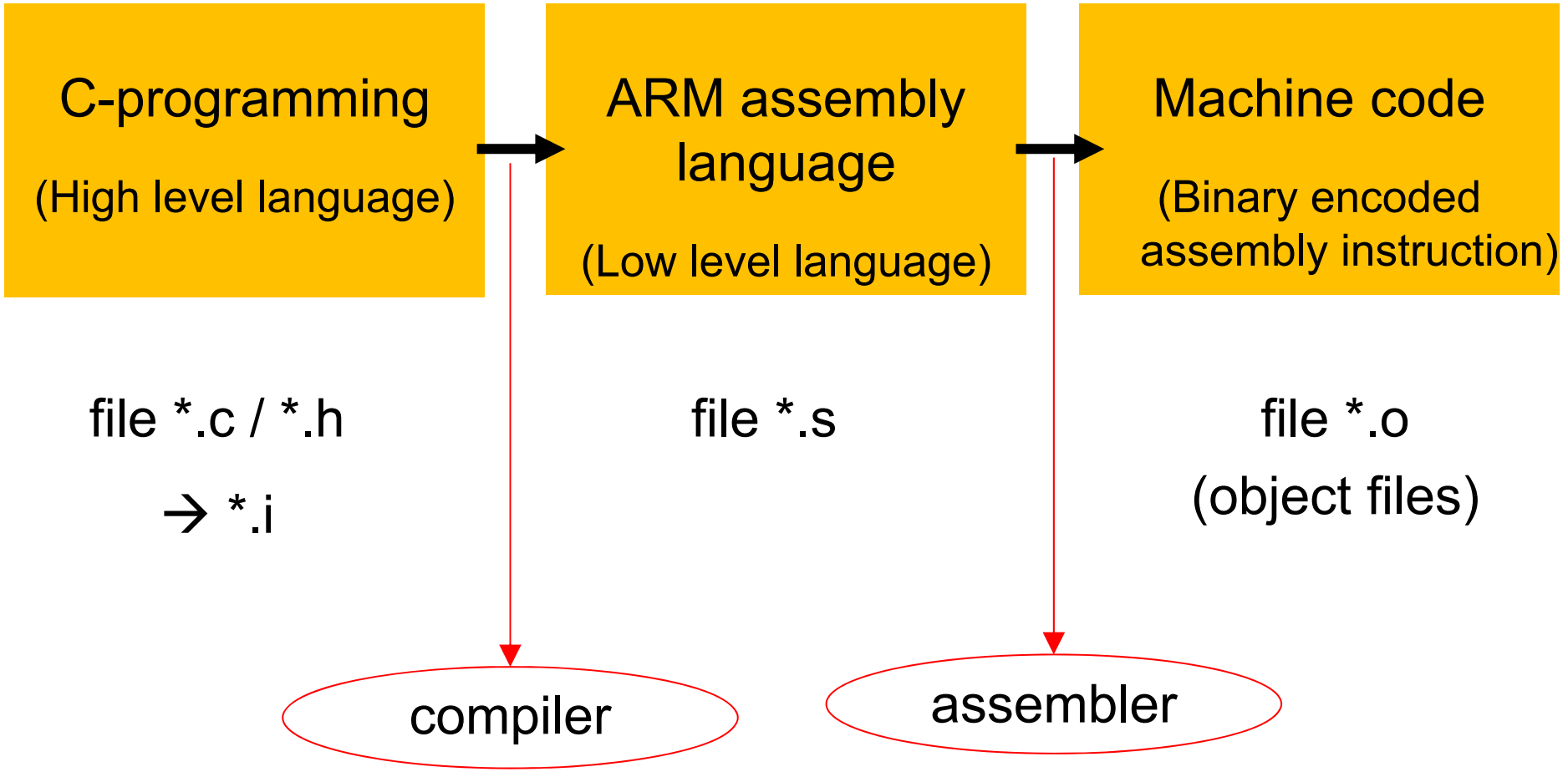
- Adds options to compiler for specific functions

```
#pragma GCC push_option
```

- Compile a function with a specific architecture

```
#pragma GCC target (“arch=armv6”)
```

Translation of programs



Linker and locator

Linker:

Combines all the object files into a single executable file
(object code uses symbols to reference other functions/variables which are defined in another object file → symbols have to be tracked and resolved).

→ relocatable file = one single object file (functions and variables are represented by symbols)

Locator:

Removes symbols and assign memory addresses into the object code.
(functions and variables names are replaced with memory addresses).

→ executable file = machine code (binary encoded instructions) that the processor understand
Format: e.g. ELF = Executable and Linkable Format