Microprocessor Concepts

Reduced Instruction Set Computers vs Complex Instruction Set Computers

RISC (e.g. ARM, Apple, Samsung)	CISC (e.g. Intel, AMD, Motorola)
Simple Instruction Type; Reduced Set (30-40)	Complex; Extended Set (100-200)
Fixed Instruction Length: One Word	Variable Instruction Length: Multi-Word
One Cycle/Instruction (Except Load & Store)	Multiple Cycles/Instruction (4-120)
Instructions Accessing Memory: Load & Store	Almost All Instructions From The Set
Arithmetic/Logic Operands Must Be In Registers	Allow Direct Memory Operations
Limited Addressing Mode	Compound Addressing Mode
Highly Pipelined – Pipelining Is Easy	Less Pipelined – Difficult
Software-Centric	Hardware-Centric; Complex Processor

Data Representation

Nibble	4 bits
Byte	8 bits
Word*	16-64 bits **32bits for STM32 32-bit ARM**

**Half of it is known as halfword, architecture dependent*

Number Systems

Binary	0b	Base-2
Octal	0o	Base-8
Hexadecimal	0x	Base-16

Decimal Conversion Examples

To convert from decimal to hex, keep dividing by the base number. E.g for 2028(DEC) to HEX, 2028/16 = 126 R 12(C). 126/16 = 7 R14(E). 7/16 = 0 R7(7) Hence, 0x7EC

From hex to dec, multiply each bit its place value and sum the products

4-bit Binary e.g.	MSB			LSB
Place Value	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
Binary Number (0b)	**1**	**0**	**1**	**1**
Product	8	0	2	1
Decimal Value (0d) = Sum of Products	11			

4-bit Octal e.g.	MSB			LSB
Place Value	$8^3 = 512$	$8^2 = 64$	$8^1 = 8$	$8^0 = 1$
Octal Number (0o)	**3**	**7**	**4**	**6**
Product	1536	448	32	6
Decimal Value (0d) = Sum of Products	2022			

4-bit Hexadecimal e.g.	MSB			LSB
Place Value	$16^3 = 4096$	$16^2 = 256$	$16^1 = 16$	$16^0 = 1$
Hexa Number (0x)	**0**	**7**	**E**	**C**
Product	0	1792	224	12
Decimal Value (0d) = Sum of Products	2028			

Decimal Base-10	Binary Base-2	Octal Base-8	Hexa Decimal Base-16
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Two's Complement

Given a n-bit binary system the range is:

Unsigned Integers	$[0, 2^n - 1]$
Signed Integers*	$[-2^{(n-1)}, 2^{(n-1)} - 1]$

**Negative Numbers have a 1 in their MSB*

**For hexadecimal F in the MSB is negative.*

Obtaining Two's Complement

*From **right to left**, starting from the LSB, look for the **first '1'** . Keep it and invert every bit to the left.*

Two's Complement of a 4 Bit System

0111	7	1111	-1
0110	6	1110	-2
0101	5	1101	-3
0100	4	1100	-4
0011	3	1011	-5
0010	2	1010	-6
0001	1	1001	-7
0000	0	1000	-8

Processor Components

Arithmetic & Logic Unit (ALU)	For performing arithmetic & logic operations
Timing and Control Unit (CU)	For fetching program instructions, decoding & executing them in sequence & transferring the results back to memory
Registers	Fast storage, where each register holds one word of data

Registers

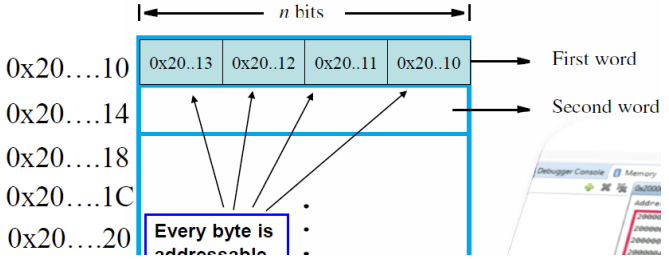
General Purpose Registers: R0 – R12 (Data & Addresses)

Program Counter (PC - R15)	Memory address of the current/next instruction
Instruction Register (IR)	Current Instruction
Link Register (LR - R14)	Return address when a subroutine/function is called
Stack Pointer (SP - R13)	Memory address of the most recent data in stack memory

* In ARMv7E-M, there are only 16 accessible registers, R0 to R15.

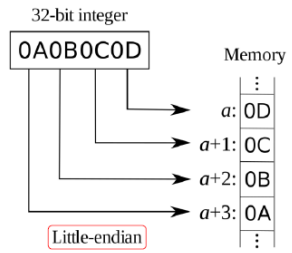
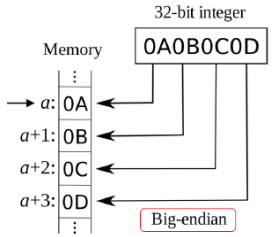
Byte Addressability

An address is addressed to each byte. Each word is 32 bits.



Endianness

Internal ordering of a sequence of bytes in computer memory, which byte is placed into the lowest address byte.

Little Endian: LSB First(32-bit ARM)	Big Endian: MSB First
	

For 32-bit arm context:

Control access & instruction fetches are always little endian, but data access can be set as either big or little endian.

Memory Opearations

CU initiates transfer of both instructions and data between memory & processor registers.

Load & Store Architecture

Only LDR and STR have access to memory operations, operands for arithmetic and logic instructions must either be in registers or hard-coded.

RISC Execution

1. Address for first instruction placed in PC
2. CU **fetches** and **executes** instructions

Fetch	Execute
CU reads value in PC for address of current instruction and places the instruction into IR	CU decodes instruction in IR and performs operation. PC is then incremented to point to next instruction

3. PC register incremented by 4 by default unless told otherwise via branching and looping

Arm Assembly Language

Calling Assembler Function from C

- Call assembler function from C program `extern int my_asm_func(int x, int y);` with max 4 input parameters(`R0`, `R1`, `R2`, `R3`) and 1 output parameter(`R0`).
- Define assembler function in asm(.s) program and use `BX LR` to return to C Program.

Glossary Terms

Optional terms: Items bracketed by { }

Op2: Flexible second source operand that can either be a constant (e.g #0xFF), or a register with optional shift. (e.g `R2, LSL #0x04`)

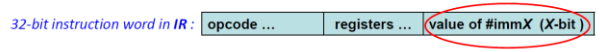
Prefix S- or U-: Signed or unsigned operation of instruction.

Suffix -S: Updates condition code flags according to result.

*S/U MUST be specified for division

Label: symbolic representations of address

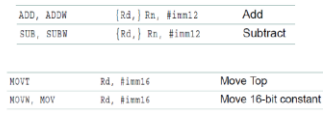
➢ **Immediate operands** – explicitly hardcoded (**#**constants)



➢ #imm8
- 2^8 Range: 0 to 255 or -128 to 127

➢ #imm12
- 2^{12} Range: 0 to 4095 or -2048 to 2047

➢ #imm16
- 2^{16} Range: 0 to 65535 or -32768 to 32767



Condition Code Flags

N	1 if result < 0, else cleared to zero
Z	1 if result $= 0$, else cleared to zero
C	For **U** operations, set if: - Result of addition $\geq 2^{32}$ - Result of subtraction ≥ 0
V	For **S** operations, set if: - Result of addition, subtraction, or compare: result $\geq 2^{31}$ or result $\leq -2^{31}$

Memory Allocation

Constants:

		Assembler Directive
.equ	Sets value of symbol to expression like `#define`	Start
.word	Allocates word-sized amount of storage space in that memory location	End

```
e.g.    .equ    STACK_TOP, 0x20008000
        .equ    PI, 314      @ .equ sets value of PI to 314
NUM1:   .word   123, 456     @ single/multiple values, e.g. array
POINTER:.word   NUM1+4       @ useful for accessing an array
```

Static Variables

e.g. .lcomm ANSWER 4 @ reserves 4 **bytes** (i.e. 1 word)

Reserves 4 bytes, that retains value globally. Lifetime is for the entire run of the program.

*All directives do not require a # to specify it's a constant.

Offset Addressing(Standard)

LDR or **STR, Offset addressing** :
LDR/STR ***Rt, [Rn {, #offset}]***

Square brackets are compulsory! They indicate **memory access**. ***Rn*** **{, #offset}** is a **pointer.**

EA: Rn + #offset

No change to Rn

Offset Addressing – Pre/Post Index

LDR/STR ***Rt, [Rn, #offset]!*** @ pre-indexed addressing
LDR/STR ***Rt, [Rn], #offset*** @ post-indexed addressing

Pre-Index	EA: Rn+#offset Rn updated to be (Rn + #offset)
Post-Index	EA: Rn Rn updated to be (Rn + #offset)

PC-Relative Addressing (LDR ONLY)

LDR ***Rd, ITEM***

performs

Rd ← [PC + offset]

ITEM is a hardcoded label of range [-4095,4095]. LDR loads Rd with the value from a **PC-relative memory address (Base**

register is always PC) specified by label, with offset calculated by the assembler.

Pseudo-Instruction (LDR ONLY)

➢ E.g. 1, loading a **32-bit** value:

```
LDR     R1, =0xA123B456     (programmer: instruction)
```

Remember the **equal sign** for pseudo-instruction!

is implemented with:

```
        LDR     R1, MEMLOC      (assembler: instruction)
MEMLOC: .word   0xA123B456      (assembler: data)
```

For loading 17–32-bit values/memory addresses. Assembler converts it to PC-relative LDR, allocates 1 word of memory fills it up with the value and lets the label(MEMLOC) refer to its address.

E.g. 2, loading an **address** (represented by a label):

```
LDR     R3, =NUM1       (programmer: instruction)
```

is implemented by the combination of:

An instruction similar to "PC-relative addressing":

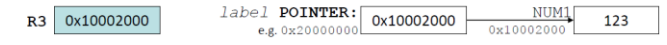
```
LDR     R3, POINTER     (assembler: instruction)
```

& a data declaration:

```
POINTER:  .word   NUM1    (assembler: data)
```

```
:e:  NUM1:    .word   123     (programmer: data)
```

Effectively, *R3 contains the **address** of NUM1; a **pointer** to 123*



Move Instructions

➢ **MOV**

➢ Assembly language format:

MOV{S} *Rd, Op2*

MOV{S} *Rd, #imm16*

```
e.g.
ADD R0, R1, #0xFF
ADD R0, R1, R2
ADD R0, R1, R2, LSL #0x4
```

Operand2

To transfer to a register an immediate constant value or from another register. Only N and Z flags are updated.

Arithmetic Instructions

Add and Subtract

➢ **ADD & SUB** (Add & Subtract)

➢ Assembly language format:

ADD{S}/SUB{S} {Rd,} Rn, Op2

or

ADD{S}/SUB{S} {Rd,} Rn, #imm12

If *Rd* is omitted, destination register is *Rn*

```
e.g.
ADD R0, R1, #0xFF
ADD R0, R1, R2
ADD R0, R1, R2, LSL #0x4
```

Operand2

**Arithmetic operations can only be done through REGISTERS.*

Hence for ADD R2, R3, R4, the procedure is:

```
LOAD R2, A
LOAD R3, B
ADD, R4, R3, R2
```

Multiply & Multiple with Accumulate

➢ Assembly language format:

MUL{S} {Rd,} Rn, Rm If *Rd* is omitted, destination register is *Rn*

MLA{S} Rd, Rn, Rm, Ra

Example:

```
MUL   R0, R1, R2      while      MLA   R0, R4, R5, R6
```

performs $R0 \leftarrow R1 \times R2$

performs $R0 \leftarrow (R4 \times R5) + R6$

Note that both MUL and MLA only return the low-order 32 bits of the 64-bit product. No dependence on signed or unsigned calculations.

To get the 64-bit products use the long versions that come with the L Suffix with the unsigned(U) or singed(S) variants:

- UMULL, UMLAL, SMULL, SMLAL

Division

➢ Assembly language format:

SDIV {Rd,} Rn, Rm *or* **UDIV {Rd,} Rn, Rm** **@ Rd ← Rn / Rm**

Note that is no DIV. The sign of the operation MUST be specified.

Compare Instructions

➢ Assembly language format:

CMP *Rn, Op2*

CMN *Rn, Op2*

If **S** suffix option is **not available** to the instruction before a conditional branch, it is very useful to **CMP/CMN** !

➢ CMP performs: Rn - Op2 *while* CMN performs: Rn + Op2

NCSV flags are updated as a result.

Branch Instructions

L: Branch with link, writes the address of the next instruction in the PC to the LR,

X: Branch indirect, branch via a register that indicates the address to branch to.

B{cond} label

The only conditional instruction that can be either inside or outside an IT block. All other branch instructions must be **unconditioned outside** an IT block and **conditional inside** the LT block.

➢ Example:

```
        CMP     R0, R1
        BEQ     IFEQUAL
        @ Some instructions for R0, R1 not equal
                :
        B       SKIPEQUAL
IFEQUAL:@ Some instructions for R0, R1 equal
                :
SKIPEQUAL: @ Continue onto the rest of the program
```

B{cond} label	$PC \leftarrow label$
BL{cond} label	$LR \leftarrow PC;\ PC \leftarrow label$
BLX{cond} Rm	$LR \leftarrow PC;\ PC \leftarrow \mathrm{Rm}$
BX{cond} Rm	$PC \leftarrow \mathrm{Rm}$

BX LR

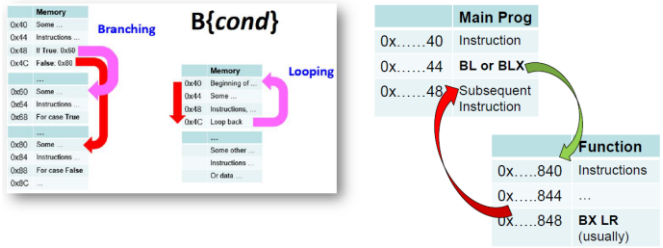
Use BX LR to branch back to the instruction mentioned. E.g

```
Main program
    ...
; R0 = X, R1 = Y, R2 = Z
BL    function 1

; Back to main program
; R0 = X, R1 = Y, R2 = Z
... ; next instructions
```

Subroutine

```
function 1
        PUSH    {R0-R2} ; Store R0, R1, R2 to stack
        ... ; Executing task (R0, R1 and R2
          ; could be changed)
        POP     {R0-R2} ; restore R0, R1, R2
        BX      LR    ; Return
```

➢ ***When to use what?*** *Usually, in situations similar to:*
- ✓ If-Else/Switch-Case branch; For/While loop: **B{*cond*}**
- ✓ To jump to a Subroutine/Function from Main: **BL** or **BLX**
- ✓ To go back to Main, from a Function: **BX**



Conditional Execution

1. Perform comparison/test or arithmetic/logical/move with suffix S to update conditions flags (NZCV)
2. Use condition code suffixes in branch instruction/IT block to perform conditional execution

Mnemonic extension	Meaning, integer arithmetic	Meaning, floating-point arithmetic[a]	Condition flags
EQ	Equal	Equal	Z == 1
NE	Not equal	Not equal, or unordered	Z == 0
CS [b]	Carry set	Greater than, equal, or unordered	C == 1
CC [c]	Carry clear	Less than	C == 0
MI	Minus, negative	Less than	N == 1
PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
VS	Overflow	Unordered	V == 1
VC	No overflow	Not unordered	V == 0
HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
GE	Signed greater than or equal	Greater than or equal	N == V
LT	Signed less than	Less than, or unordered	N != V
GT	Signed greater than	Greater than	Z == 0 and N == V
LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
None (AL) [d]	Always (unconditional)	Always (unconditional)	Any

IF-THEN block

Maximum 4 instructions and must start with T. So maximum is `IT<x><y><z> <cond>`. E.g

```
CMP R0, R1        ; Compare R0 and R1
ITTEE EQ          ; If R0 equal R1, Then-Then-Else-Else
ADDEQ R3, R4, R5  ; Add if equal
ASREQ R3, R3, #1  ; Arithmetic shift right if equal
ADDNE R3, R6, R7  ; Add if not equal
ASRNE R3, R3, #1  ; Arithmetic shift right if not equal
```

Logic Instructions

➢ **AND, ORR & EOR** (***bit-wise*** logic AND, OR & Exclusive-OR)

➢ Assembly language format:

op{*S*} **{*Rd,*} *Rn, Op2***

where op is one of the above

Recall: *Operand2*

```
ADD R0, R1, #0xFF
ADD R0, R1, R2
ADD R0, R1, R2, LSL #0x4
```

➢ Example:

ANDS Rd, Rn, Rm

performs the bit-wise logic AND of the operands in registers Rn & Rm, writes the result into register Rd. N & Z are updated accordingly, C may be updated based on the result of Op2 (e.g. LSL #0x4, see Shift), V is not updated in all logic instructions

MVN (Move NOT: ***bit-wise*** logic NOT)

Assembly language format:

MVN{*S*} ***Rd, Op2***

performs a bit-wise logic NOT operation on the value of *Op2*, & places the result into Rd.

Rd ← (~Op2)

Example:

MVNS Rd, Rn

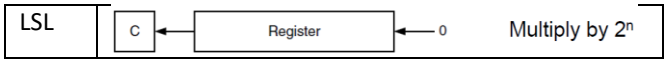
performs the bit-wise logic NOT on Rn, writes the result into register Rd. N & Z are updated accordingly, C may be updated based on the result of Op2 (e.g. LSL #0x4), V is not updated in all logic instructions

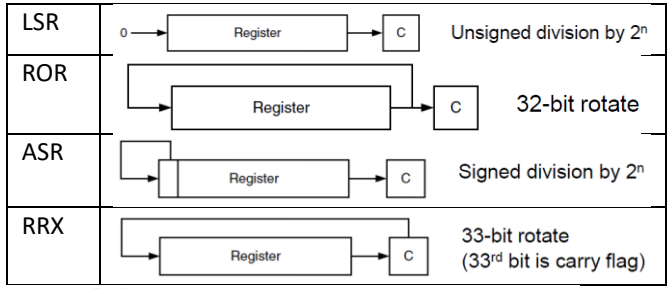
Shift and Rotate

LSL Logical Shift Left
LSR Logical Shift Right
ASR Arithmetic Shift Right
ROR Rotate Right
RRX Rotate Right with Extend

op{*S*} ***Rd, Rm, Rs***
op{*S*} ***Rd, Rm, #n***
where *op* is one of the top 4, Rs & *n* (shift length) limited to =<31/32, &
RRX{*S*} ***Rd, Rm***

Note that S suffix should be used to update the carry flag.





5.3 Shift & Rotate Instructions: E.g.

➢ Inline Barrel Shifter Example, recall Op2:

MOVS R0, R1, **ASR** #2 @ R0 ← R1>>2 (i.e. Op2)

If R1 = 0xA0000014 $(1010\ldots.\ \ldots.01\ 0100)_2$
R1>>2 = 0xE8000005 $(1110\ 10..\ \ldots.00\ 0101)_2$

which results in 0xE8000005 being written to R0

➢ The most significant bit (sign bit) of R1 is being copied in every shift, i.e. for ***n*** times

➢ Carry flag is finally cleared, due to the 0 in the original n^{th} position (***n***=2 in this ASR) of R1 before the shift

Test Instructions

TST ***Rn, Op2***
TEQ ***Rn, Op2***

TST: Bitwise logical AND
TEQ: Bitwise logical Exclusive OR
Both update N & Z, C might be updated based on Op2. Note that the result is discarded unlike ANDS and EORS.

Examples:

TST R3, #1 @ bit-wise logic AND

sets Z = 1 if least significant bit of R3 is 0
sets Z = 0 if least significant bit of R3 is 1
(useful for checking status bits in I/O devices)

TEQ R2, #5 @ bit-wise logic Exclusive OR

sets Z = 1 if R2 equals 5
sets Z = 0 otherwise
(also useful for testing the sign of a value & check N flag)

Stack and Subroutine Functions

Data saved in a last-in-first-out manner and address specified by SP. Different stack implementations:

Empty	Full
SP points to the next **free location** in the stack	SP points to the **most recent item** in the stack

Ascending	Descending
Starts from low and goes to higher memory addresses as items are **pushed** onto the stack	Starts from high and goes to lower memory addresses as items are **pushed**.

By default, Cortex-M4 stack is **full descending,** with the common use to save register contents before data processing & then restore those contents when done.

PUSH at start, and then POP at the end.

➢ E.g.: **PUSH {R0}**

performs
R13 ← R13 - 4
followed by
Memory [R13] ← R0

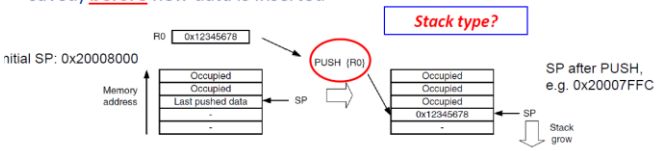
POP {R0}

performs
R0 ← *Memory* [R13]
followed by
R13 ← R13 + 4

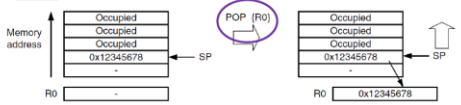
Note that curly braces here are not optional, and an array of registers can be pushed or popped. The order does not matter, and the lowest numbered registers from lowest memory address.

E.g. initial SP set to point at: 0x20008000 (always empty)

- SP (R13) always points to the last data pushed into stack memory
- When **PUSH**, SP **decrements** by 4 or multiples of 4 (if several registers are saved) **before** new data is inserted

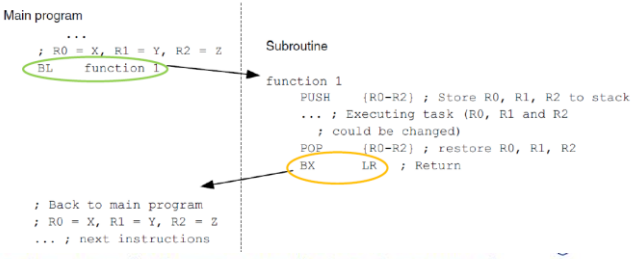


- When **POP**, SP **increments** by 4 or multiples of 4 (if several registers are recalled) **after** data is copied *(What happen to stack's contents after POP?)*

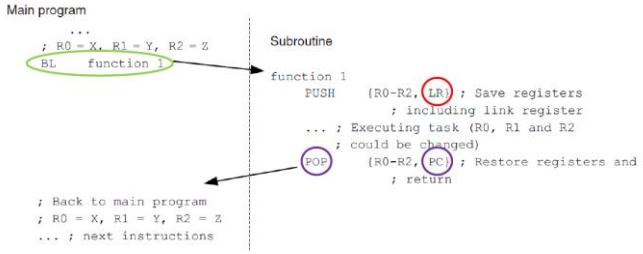


Subroutines *(software-controlled stack operation)*:
Manually PUSH **registers** that will be modified in the subroutine, then POP them at the end of the subroutine to restore their original contents

Multiple-registers PUSH & POP are similar to multiple-word STR & LDR respectively (*refer to STM & LDM for details, not in syllabus*)



Subroutines *(software-controlled stack operation)*:
Manually PUSH **registers** that will be modified & **LR** in the subroutine, then POP registers at the end to restore their original contents & **LR into PC**, thus **combining POP & BX LR** into one instruction.



Tutorial 2 Questions

Determining the number of negative numbers in a list

```
        LDR    R1, =NUMBERS      @ Load address of data list
        LDR    R2, N             @ Load size of list
        MOV    R0, #0            @ Clear negative number counter

LOOP:   LDR    R3, [R1], #4      @ Load next data item
        CMP    R3, #0            @ Set condition code flags
        IT     MI                @ Start of IT block
        ADDMI  R0, #1            @ Increment counter if data
                                 item is negative
        SUBS   R2, #1            @ Decrement loop counter
        BNE    LOOP              @ Branch back if not done
        LDR    R4, =NEGNUM
        STR    R0, [R4]          @ Store result
```

Reverse the order of bits in register R2:

```
        MOV    R0, #32           @ Load R0 with count value 32
LOOP:   LSLS   R2, R2, #1        @ Shift contents of R2 left one
                                 bit position, setting the high-
                                 order bit into the C flag
        RRX    R1, R1            @ Rotate R1 right one bit
                                 position, including the C flag
        SUBS   R0, #1            @ Check if finished
        BNE    LOOP
        MOV    R2, R1            @ Load reversed pattern back
                                 into R2
```

Generate the first n numbers of the Fibonacci Series

```
        LDR    R0, N             @ Use R0 for loop count for
                                 numbers generated after 1
        SUB    R0, #2
        LDR    R1, =MEMLOC       @ Use R1 as memory pointer
        MOV    R2, #0            @ Store first two numbers
        STR    R2, [R1], #4      @ 0 and 1, from R2
        MOV    R3, #1
        STR    R3, [R1], #4      @ and R3, into memory

LOOP:   ADD    R3, R2, R3        @ Starting with number i - 1 in
                                 R2 and i in R3, compute and
                                 place i + 1 in R3
        STR    R3, [R1], #4      @ and store in memory

        SUB    R2, R3, R2        @ Recover old i and place in R2

        SUBS   R0, #1            @ Branch back if all numbers
                                 have not been computed
        BGT    LOOP
```

Assignment 1 Insertion Sort

```
@ R0 ...Input: (int*)arr (Mem address) -> should be output return value (n
@ R1 ...Input: (int)M (size of the array)
@ R2 mem add of curr pos in arr
@ R3 mem add of curr item (for loop switching)
@ R4 value of original item
@ R5 value of comparable item
@ R6 mem add of arr[0] , exit condition for stopping iter

@ write your program from here:
insertion_sort:
    PUSH {R2-R6, R14}

    MOV R2, R0       //Arr pos
    MOV R3, R0       //Cur pos
    MOV R6, R0       //Exit cond
    MOV R0, #0       //Set no of swaps to 0
    B SORT

SORT:
    CMP R1, #1       //Exit if all items have been iterated though
    BLS EXIT

    MOV R3, R2       //Return swap pos back to orig array pos
    SUB R1, #1
    B SWAP

SWAP:
    LDR R4, [R3]      //Load values in arr into R4, R5
    LDR R5, [R3,#4]
    CMP R4, R5        //Compare values and swap them in memory if needed
    ITTTE GT
    ADDGT R0, #1
    STRGT R5, [R3]
    STRGT R4, [R3,#4]
    BLE  EXITSWAP     //Else exit swap, and move on to next item in arr

    CMP R3, R6        //also exit if swapping has reached idx 0 of arr
    BEQ EXITSWAP

    SUB R3, #4        //Iteratively do the swap check on idx - 1
    B SWAP

EXITSWAP:
    ADD R2, #4        //Increment idx by 1 and cont insertions sort
    B SORT

EXIT:
    POP {R2-R6, R14}   //register range must be in ascending order?
    BX LR
```

C Programming

Data Types

Built-in/Basic*	Derived	User-defined
Void, char, short, int long, float, double	Array, pointer, references	Struct, union, enum

*Note that built-in data types are OS/compiler dependent.
The minimum of a unsigned int is either 2 or 4 bytes.
Data types in stdint.h have a guaranteed, fixed data size.
E.g int8_t, uint8_t, int16_t, uint16_t, … int64_t, uint64_t

Type Conversion

Implicit(Automatic Type Conversion)

Compiler automatically promotes the lower type to the higher type to avoid data loss. However, data loss is still possible (int > unsigned)

bool -> char -> short -> unsigned short -> int -> unsigned -> long -> unsigned long -> long long -> unsigned long long -> float -> double …

Explicit(Type Casting)

C syntax: (type)expression
Should be avoided as the result can be unpredictable.

Variables & Constants

- Only alphanumeric characters & underscore are allowed.
- Cannot begin with a number. mdb1 is okay, 1mdb isn't.
- Names beginning with '_' are meant for reserved identifiers while a trailing '_t' are reserved for standard types.

auto	float	signed	_Alignas (since C11)
break	for	sizeof	_Alignof (since C11)
case	goto	static	_Atomic (since C11)
char	if	struct	_Bool (since C99)
const	inline (since C99)	switch	_Complex (since C99)
continue	int	typedef	_Decimal128 (since C23)
default	long	union	_Decimal32 (since C23)
do	register	unsigned	_Decimal64 (since C23)
double	restrict (since C99)	void	_Generic (since C11)
else	return	volatile	_Imaginary (since C99)
enum	short	while	_Noreturn (since C11)
extern			_Static_assert (since C11)
			_Thread_local (since C11)

C keywords/reserved words are not allowed!
Note that #define is used to declare constants, but it **cannot be type checked**.

Advanced Operators

sizeof()
Ternary(?:)
<condition>?<if true instruction>:<else instruction>

Operator Precedence

Operator	*Description*	*Associativity*
() [] . ->>> ++ --	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left
* / %	Multiplication, division and modulus	left to right
+ -	Addition and subtraction	left to right
<< >>	Bitwise left shift and right shift	left to right
< <= > >=	relational less than/less than equal to relational greater than/greater than or equal to	left to right
== !=	Relational equal to and not equal to	left to right
&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
|	Bitwise inclusive OR	left to right
&&	Logical AND	left to right
||	Logical OR	left to right
? :	Ternary operator	right to left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment operator Addition/subtraction assignment Multiplication/division assignment Modulus and bitwise assignment Bitwise exclusive/inclusive OR assignment	right to left
,	Comma operator	left to right

Data types & size

char	1 byte
float	4 bytes
int	4 bytes

Selection

1. if-else

```
if (condition 1) {
  // code block to be executed if condition 1 is true
} else if (condition 2) {
  // code block to be executed if the condition 1 is false & condition 2 is true
} else {
  // code block to be executed if the condition 1 is false & condition 2 is false
}
```

2. switch-case

```
Syntax:
switch(expression) {
  case x:
    // code block to be executed if x is true
    break;
      /* break ends this case & passes the control to the next statement
         that is immediately after the switch-case construct */
  case y:
    // code block to be executed if y is true
    break;
  default:
    // code block to be executed if there is no case match
}
```

Iteration

For, while, do-while. Break vs continue.

```
Syntax:
for (initial condition; test; update) {
  // code block to be executed for as many times as the test result is true
}
while (test) {
  // code block to be executed while test result is true
}
do {
  // code block to be executed at least once first & while test result is true
    } while (test);
```

Functions

Pass by value: value of parameter passed into the function. Value can be used but not changed.
Pass by reference: address of parameter passed into function. Value can be used and change. Format: `int fx (int *a, char *b) {`

Function-like macros

Syntax: *#define macro_name(arg) (macro_definition)*
E.g. 1: #define incrBy2(x) x+2 //brackets for definition is optional

Printf()

Syntax: `printf("format ", arg1, arg2);`

- Format specifiers:
  - %d decimal integer
  - %u unsigned decimal integer
  - %o octal integer
  - %x hexadecimal integer
  - %c single character
  - %s character string
  - %f floating point number
  - %e exponential form
  - %p pointer (i.e. an address)

Note: %d, u, o, x, f & e may have l prefix for long

Arrays

1D: `int x[100];`

2D: `float mydata[8][25];`

```
int y[3][4] = {  {1,2,3,4},
                 {5,6,7,8},
                 {9,10,11,12}
              };
```

Addr

Structures

General

```
struct students {
   char name[30]; int phone;
};
struct students st1;
st1.phone = 20;
```

Typedef

```
typedef struct ece /* the name ece is
optional */{
   char name[30]; int phone;
} students ; /* students is a "struct
ece " variable */
students st1; /* instead of struct
students
```

Pointers

Pointer is a variable that stores the address of another variable.

&: Address Operator that evaluates the memory address of the operand

*: Indirection operator that evaluates the value of its pointer variable operand. Only operates on pointers.

```
int x = 10;
int *ptr = &x
int res = *ptr; // which will be 10
```

Pointer to Structures

Pass address of struct to function instead of entire struct

- The time needed to perform the pass is independent of struct size.
- Allows the function to directly modify the struct and avoid making a copy of the struct variable.
- Allow efficient management of dynamically-memory-allocated structures.
- Anable access to variables defined outside of function.
- Enable easy return of multiple values from function.

```
typedef struct student{
   int id;
   char name[30];
}student_record;

void print_rec(student_record *X){
   printf("ID is :%d \n", X->id);
}

void update_id(student_record *X)
   X->id = 10;
}

student_record sr;
student_record *ptr_sr = &sr;
update_id(ptr_sr);
```

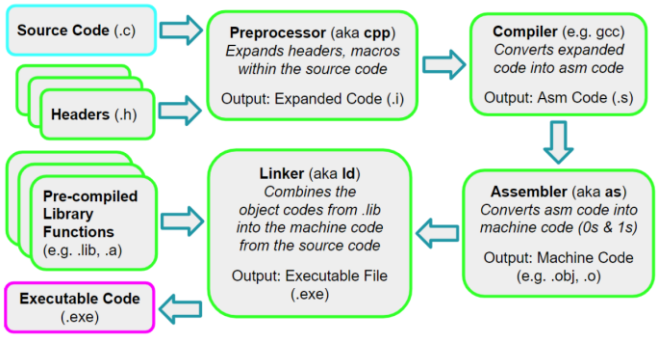
Note that `Ptr->member` is short for `(*Ptr).member` and is used to access data members in struct pointers.

```
int main()
{
    student_record student1 = {1, "Adam", 90.5};
    student_record *sptr; // declare sptr as a PtS

    sptr = &student1; // "&" operator gets the address of the data object

    print_rec(sptr); // print structure members via the PtS
    update_score(sptr, 98.7); // updates "score" member to 98.7 via the PtS
    print_rec(sptr);

    return 0;
}

void print_rec(student_record *X) // *X is a PtS of student_record
{
    printf("Record of STUDENT: \n");
    printf("  Id is: %d \n", X->id);
    printf("  Name is: %s \n", X->name);
    printf("  Score is: %f \n", X->score);
    printf("\n");
}

void update_score(student_record *X, float newscore)
{
    X->score = newscore;
}
```

Output:

```
Record of STUDENT:
  Id is: 1
  Name is: Adam
  Score is: 90.500000

Record of STUDENT:
  Id is: 1
  Name is: Adam
  Score is: 98.700000
```

Compilation Process



1. Preprocessor($.c\,.h \rightarrow .i$)
2. Compiler($.i \rightarrow .s$)

- Note that assembly is specific to target processor, and the code can differ even if the same source code and compiler are used.

3. Assembler($.s \rightarrow .obj/.o$)
4. Linker ($.obj/.o,\ .lib/.a \rightarrow .exe$)

Tutorial 3 Questions

Reverse order of 8 bits of a character and add 30:

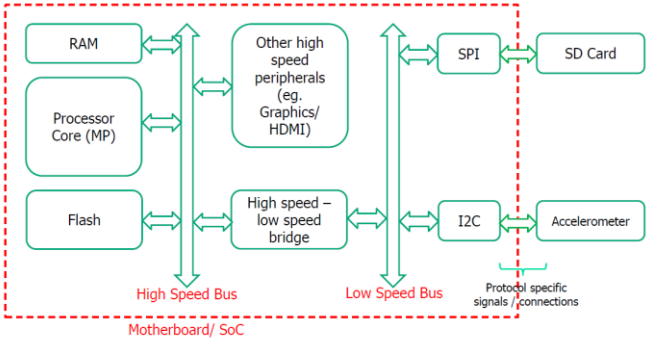
```c
int main() {

/* Reverse the original bit order by:
      first swapping the top and bottom nibbles (4 bits),
      then the top and bottom crumbs (2 bits) within each nibble,
      and finally, between every pair of adjacent bits, see Fig 1 */

    char toswap = 'T';
    char swapped;

      // swap nibbles - 4 bit groups
    toswap = (toswap & 0b11110000)>>4 | (toswap & 0b00001111)<<4;
      // toswap = (toswap & 0xF0)>>4 | (toswap & 0x0F)<<4; // also ok

      // swap crumbs - 2 bit groups
    toswap = (toswap & 0b11001100)>>2 | (toswap & 0b00110011)<<2;
      // toswap = (toswap & 0xCC)>>2 | (toswap & 0x33)<<2; // also ok

      // swap bits
    swapped = (toswap & 0b10101010)>>1 | (toswap & 0b01010101)<<1;
      // swapped = (toswap & 0xAA)>>1 | (toswap & 0x55)<<1; // also ok

    printf("%c\n",swapped); // will print *

    printf("%c\n",swapped+30); // will print H

    return 0;
}
```

Fibonacci using Pointers:

```c
void fibonacci(int n, int *fibo)
{
     *(fibo) = 0;
     *(fibo+1) = 1;
     for(int i=2; i<n; i++)
     {
          *(fibo+i) = *(fibo+i-1)+*(fibo+i-2);
     }
     return;
}
```

Interfacing Concepts

Multi-tiered Bus Architecture



Peripheral Registers

Accessed like memory using LDR/STR equivalent.

Status Registers: read by processor to know the status of hardware.

Control Registers: written by processor to give commands or control signals.

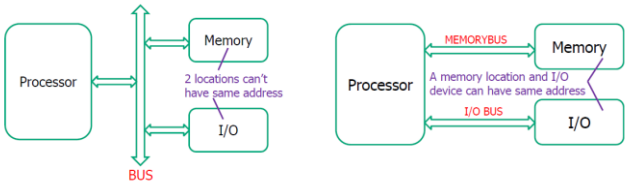
Data Registers: read(for input) and/or written(for output) to perform actual transfer of data.

Abstractions

CMSIS	Core Peripherals (NVIC, interrupts, SysTick)
HAL	On-chip Peripherals(GPIO, I2C, SPI, UART)
BSP	External Devices on baseboard(sensors)

Access

Memory-mapped IO(MMIO): Address for peripheral registers shares the address space with the memory (LDR/STR used)
Port-mapped IO(PMIO) Memory and peripherals are in separate address spaces, with separate instructions.



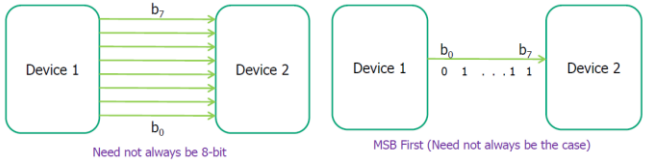
Protocols

Parallel vs Serial

Serial(SATA, I2C, SPI, UART)

Parallel(PCI, IDE, Printer Port)

- High speed low distance
- Suffers from bus skew (different signals, different speeds) and cross talk (signal interference)



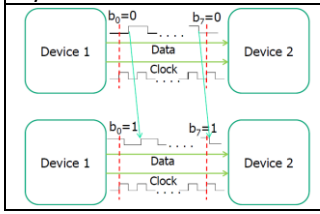
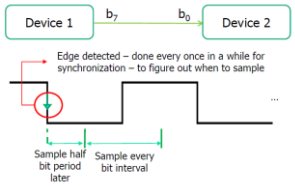
Synchronous vs Asynchronous Protocols

Synchronous(SPI, I2C)

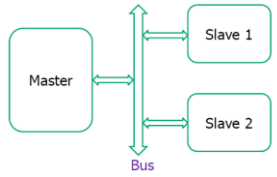
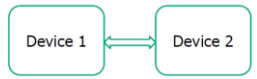
- One wire carries clock for time reference.
- Faster but issue of cross talk, bus skew

Asynchronous(UART,USB)

- Only data is transmitted, no clock.
- Receiver needs to recover timing info from data/control bits (start/stop etc.) – slower, complicated hardware.

Synchronous	Asynchronous
	

Bus-based vs Point-to-point

Bus-based: each device has a unique address (I2C, SPI, USB)	Point-to-point: two devices have a dedicated link (UART)
	

Master-slave vs Peer-to-Peer

Master-Slave(I2C, SPI, USB, AHB)

Only a master device can initiate a communicate, decides which slave should respond and may generate the clock.

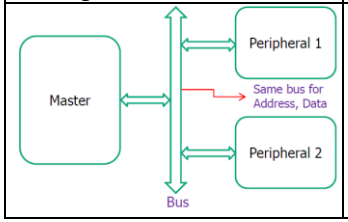
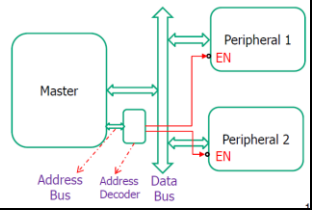
Peer-to-peer(Ethernet)

Any device can initiate communication.

Simplex vs Half Duplex vs Full Duplex

Simplex (GPIO output to LED, or input from switches)	single, unidirectional link, one-way interaction
Half Duplex (I2C, USB 2.0)	one bi-directional link, devices take turns
Full Duplex (SPI, UART, USB 3.x)	at least two links, can transmit and receive at the same time

Addressing: In band vs Out-of-band

In band (I2C, USB): address of the device being accessed, and data are sent through the same bus	Out-of-band(SPI): separate address bus, a decoder enables (activates) the device
	

GPIO

GPIO ports: GPIOA – GPIOI with up to 16 I/O pins per port. Direct connection to AHB2 bus allows fast I/O timing.

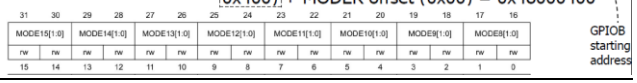
Register Type	Register Name (where x = A to H)		Read-Write
Control	Mode register	GPIOx_MODER	RW
	Output type register	GPIOx_OTYPER	RW
	Output speed register	GPIOx_OSPEEDR	RW
	Pull-up / Pull-down register	GPIOx_PUPDR	RW
	Configuration lock register	GPIOx_LCKR	RW
	Alternate function low register	GPIOx_AFRL	RW
	Alternate function high register	GPIOx_AFRH	RW
	Analog switch control register	GPIOx_ASCR	RW
Data	Input data register	GPIOx_IDR	RO
	Output data register	GPIOx_ODR	RW
	Bit reset register	GPIOx_BRR	WO
	Bit set/reset register	GPIOx_BSRR	WO

GPIO Control Register – GPIOx_MODER

Used to configure the mode of pin y (15:0) of port x(A to H)

00	Input Mode
01	GPIO Mode
10	AF mode
11	Analog Mode (reset state)

E.g. To configure Port B Pin 14 as GPIO output (x=B, y=14):
MODE**14**[1:0] of GPIO**B**_MODER, i.e., GPIO**B**_MODER [**29:28**] = **01**
GPIOB_MODER address = AHB2 base (0x48000000) + GPIOB offset (1 KB = 0x400) + MODER offset (0x00) = 0x48000400



GPIO Control Register – GPIOx_AFRL/H

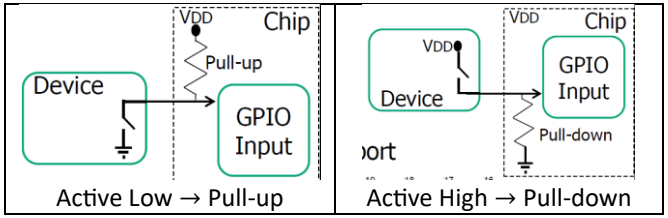
AFRL/H: Alternate Function Low/High. 16 possible functions so 4-bit value per pin. AFRL for pins 0-7 and AFRH for 8-15.
0000: AF0, 0001: AF1, 0010: AF2 … 1111:AF15

E.g., To configure Port C Pin 12 as UART5_TX (x=C, y=12): AFSEL**12**[3:0] of GPIO**C**_AFR**H**, i.e., GPIO**C**_AFR**H**[**19:16**] = 4'b**1000** (UART5_TX is AF8)

GPIO Control Register: GPIOx_PUPDR

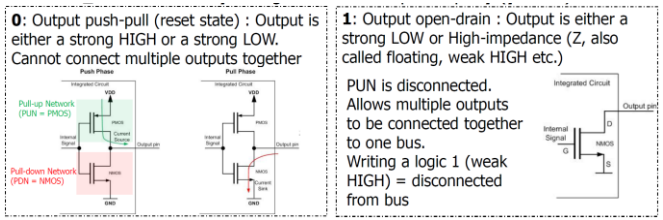
Active HIGH input – pull down. The vice versa.

00	No pull-up, no pull-down used when pin is not an input, or when external device will driving a strong high or low and won't be floating
01	Pull-up: Default input is high
10	Pull-down: Default input is low
11	Reserved

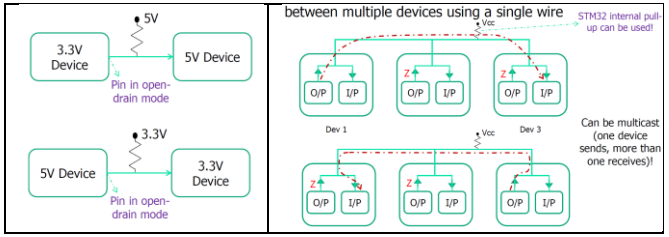


GPIO Control Register: GPIOx_OTYPER

OTYPER: Output type register



Open-drain mode can be used to interface between different logic levels or allow bi-directional communication between multiple devices with a single wire.



GPIO Data Registers

Read data from the Input data register (IDR) and write data to the ODR. But its better to manipulate ODR bits using the BSRR or BRR, which are logical interfaces for affecting ODR.

- Allows for one action modification rather than $read \rightarrow modify \rightarrow write\ back$

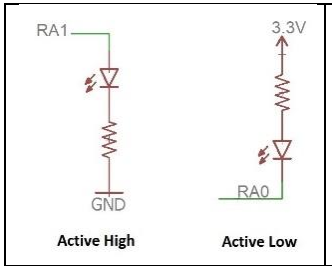
GPIOx_IDR IDy: Logic Level read from port x pin y.
GPIOx_ODR ODy: Logic level output on port x pin y. Read-write, can read the output or write to specific bit while keeping the rest unchanged.

GPIOx_BRR BRy: for port x pin y. Writing 1 to a bit will reset that pin LOW
GPIOx_BSRR: port x pin y
BSy[15:0] writing 1 will set the pin HIGH. [PRIORITY]
Bry[31:16] writing 1 will reset the pin. To be used if there is a need to set and reset some pins at the same time.

GPIO HAL Supporting Functions

```
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOB, LED2_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin LED2_Pin */
  GPIO_InitStruct.Pin = LED2_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
}
```

```
/* Initialization and de-initialization functions *****************************/
void              HAL_GPIO_Init(GPIO_TypeDef  *GPIOx, GPIO_InitTypeDef *GPIO_Init);
void              HAL_GPIO_DeInit(GPIO_TypeDef  *GPIOx, uint32_t GPIO_Pin);

/* IO operation functions *****************************************************/
GPIO_PinState     HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
void              HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState);
void              HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
HAL_StatusTypeDef HAL_GPIO_LockPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
void              HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin);
void              HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin);
```



Active high → OTYPER must be Push-pull. Write to BSRR (HIGH) to turn off LED.
Active low → doesn't matter.
Write to BRR (LOW) to turn on LED.

Bus	Boundary address	Size (bytes)	Peripheral
AHB2	0x4800 1C00 - 0x4800 1FFF	1 KB	GPIOH
	0x4800 1800 - 0x4800 1BFF	1 KB	GPIOG
	0x4800 1400 - 0x4800 17FF	1 KB	GPIOF
	0x4800 1000 - 0x4800 13FF	1 KB	GPIOE
	0x4800 0C00 - 0x4800 0FFF	1 KB	GPIOD
	0x4800 0800 - 0x4800 0BFF	1 KB	GPIOC
	0x4800 0400 - 0x4800 07FF	1 KB	GPIOB
	0x4800 0000 - 0x4800 03FF	1 KB	GPIOA

STM32L475xx devices peripheral register boundary addresses

- Header File:
  Drivers/STM32L4xx_HAL_Driver/Inc/stm32l4xx_hal_gpio.h
  - Contains declarations of constants, structures, functions etc used for GPIO peripheral

```
/* Initialization and de-initialization functions *****************************/
void               HAL_GPIO_Init(GPIO_TypeDef  *GPIOx, GPIO_InitTypeDef *GPIO_Init);
void               HAL_GPIO_DeInit(GPIO_TypeDef  *GPIOx, uint32_t GPIO_Pin);

/* IO operation functions *****************************************************/
GPIO_PinState      HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
void               HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState);
void               HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
HAL_StatusTypeDef HAL_GPIO_LockPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
void               HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin);
void               HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin);
```

- Source File:
  Drivers/STM32L4xx_HAL_Driver/Src/stm32l4xx_hal_gpio.c
  - Contains the definitions of GPIO functions

- Example: use HAL_GPIO_WritePin() to output 1 from Port B Pin 14:
- Assume the GPIO pin has already been configured as Output (by Mode Register)
- User program: HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_SET);

```
void HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState)
{
  /* Check the parameters */
  assert_param(IS_GPIO_PIN(GPIO_Pin));
  assert_param(IS_GPIO_PIN_ACTION(PinState));

  if(PinState != GPIO_PIN_RESET)
  {
    GPIOx->BSRR = (uint32_t)GPIO_Pin;
  }
  else
  {
    GPIOx->BRR = (uint32_t)GPIO_Pin;
  }
}
```

```
#define GPIO_PIN_14                ((uint16_t)0x4000)  /* Pin 14 selected */
typedef struct
{
  __IO uint32_t MODER;    /*!< GPIO port mode register,               Address offset: 0x00 */
  __IO uint32_t OTYPER;   /*!< GPIO port output type register,        Address offset: 0x04 */
  __IO uint32_t OSPEEDR;  /*!< GPIO port output speed register,       Address offset: 0x08 */
  __IO uint32_t PUPDR;    /*!< GPIO port pull-up/pull-down register,  Address offset: 0x0C */
  __IO uint32_t IDR;      /*!< GPIO port input data register,         Address offset: 0x10 */
  __IO uint32_t ODR;      /*!< GPIO port output data register,        Address offset: 0x14 */
  __IO uint32_t BSRR;     /*!< GPIO port bit set/reset  register,     Address offset: 0x18 */
  __IO uint32_t LCKR;     /*!< GPIO port configuration lock register, Address offset: 0x1C */
  __IO uint32_t AFR[2];   /*!< GPIO alternate function registers,     Address offset: 0x20-0x24 */
  __IO uint32_t BRR;      /*!< GPIO Bit Reset register,               Address offset: 0x28 */
  __IO uint32_t ASCR;     /*!< GPIO analog switch control register,   Address offset: 0x2C */
} GPIO_TypeDef;
#define GPIOB               ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOB_BASE          (AHB2PERIPH_BASE + 0x0400UL)
#define AHB2PERIPH_BASE     (PERIPH_BASE + 0x08000000UL)
#define PERIPH_BASE         (0x40000000UL) /*!< Peripheral base address */
```

```
typedef enum
{
  GPIO_PIN_RESET = 0U,
  GPIO_PIN_SET
}GPIO_PinState;
```

```
#include "main.h"
static void MX_GPIO_Init(void);

int main(void)
{

  HAL_Init();
  MX_GPIO_Init();

  while (1)
  {
      HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_14);
      HAL_Delay(100);
  }
}

static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOB, LED2_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin LED2_Pin */
  GPIO_InitStruct.Pin = LED2_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
}
```

Interpreting Sensor Output

- The raw *value* given by a sensor typically does not represent the *magnitude* of the physical quantity directly
- *magnitude* = *f*(*value*) where *f* is a conversion function which is often, but not necessarily, linear
- Many sensors give 16-bit *value*s as two separate bytes, as the protocols (e.g., I²C) may deal with data only in chunks of 8 bits
- Concatenate the bytes

value_LSB	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
value_MSB	B_{15}	B_{14}	B_{13}	B_{12}	B_{11}	B_{10}	B_9	B_8

B_{15}	B_{14}	B_{13}	B_{12}	B_{11}	B_{10}	B_9	B_8	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0

value

=

B_{15}	B_{14}	B_{13}	B_{12}	B_{11}	B_{10}	B_9	B_8	0	0	0	0	0	0	0	0

((uint16_t)value_MSB << 8)

OR, i.e., '|' can also be used instead of '+', in which case the outer () is unnecessary

+

0	0	0	0	0	0	0	0	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0

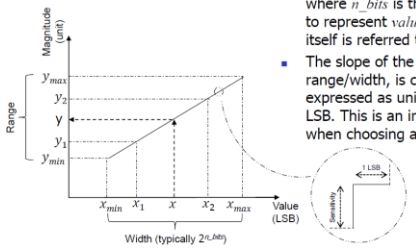
(uint16_t)value_LSB;

Interpreting Sensor Output

magnitude is obtained through a linear interpolation

Recall: Given that two points (x_1, y_1) and (x_2, y_2) lie on a line, the equation of the line is

$$y = y_1 + (x - x_1) \times \frac{y_2 - y_1}{x_2 - x_1}$$



- Often, the reference points (x_2, y_2), (x_1, y_1) are the same as the extreme points (x_{max}, y_{max}), (x_{min}, y_{min})
- $[y_{min}, y_{max})$ is the **range** of *magnitude*, often expressed as $y_{max} - y_{min}$
- $x_{max} - x_{min}$ is the **width**, typically 2^{n_bits}, where n_bits is the number of bits used to represent *value* (sometimes, n_bits itself is referred to as the width)
- The slope of the line, typically range/width, is called the **sensitivity**, expressed as unit_of_*magnitude* per LSB. This is an important consideration when choosing a sensor

- Different sensor datasheets use different terminologies and have different ways of providing the slope of the line. Some sensors allow the sensitivity to be configured (there is a tradeoff with range)
- Example: a light sensor can measure in the range [100 lux, 500 lux), giving a 16-bit unsigned value
  - In this case, $f(value) = 100 + (value - 0) * \text{sensitivity}$
  - Sensitivity = $(y_{max} - y_{min})/2^{n\_bits}$ = (float) (500-100)/0x10000 = 0.0061 lux/LSB
  - If value_MSB = 0x7B , value_LSB = 0xF3, then magnitude = 100 + (float) 0x7BF3*sensitivity = 293.67 lux

  $(x_{min}, y_{min}) = (0, 100)$ is on the line
- Example: an accelerometer gives a 16-bit signed value, with a sensitivity of 0.00122 m/s²/LSB
  - In this case, $f(value) = 0 + (value - 0) * \text{sensitivity}$
  - If value_MSB = 0x7B , value_LSB = 0xF3, then magnitude = (float) 0x7BF3*0.00122 = 38.71 m/s²
  - Range = 2^16 * sensitivity = 80 m/s², i.e., ±40 m/s² i.e., [-40, 40) m/s²
  - What if value_MSB = 0xFF , value_LSB = 0xF3? Hint: The acceleration is –ve, close to 0!

  (0, 0) is on the line though it is not (x_{min}, y_{min}) – this is an assumption you can make unless stated/implied otherwise
- Be mindful of signed vs unsigned and adjust concatenation as well as the conversion formula as appropriate

Tut 4 Solutions

Read BTN1 and turn on LED1

A push button BTN1 is connected to STM32L475VG port A pin 10. BTN1 is active low, and LED is active high. Assuming that the pins are already configured to be in the correct mode, write a C program to turn on the LED whenever BTN1 is pressed. (Direct register modification).

```
unsigned int* GPIOA_IDR = (unsigned int*) 0x48000010;
unsigned int* GPIOB_BSRR = (unsigned int*) 0x48000418;
unsigned int* GPIOB_BRR = (unsigned int*) 0x48000428;

while(1)
{
	if ( (*GPIOA_IDR >> 10) & 0x01)
		// if bit 10 of GPIOA_IDR is 1,
		// if ( *GPIOA_IDR & (1 << 10) ) also works
		// BTN1 is not pressed (BTN1 is active low)
	{
		*GPIOB_BRR = (1<<14);
		// write 1 to bit 14 of BRR to turn off LED
	}
	else // BTN1 is pressed
	{
		*GPIOB_BSRR = (1<<14);
		// write 1 to bit 14 of BSRR to turn on LED
	}
}
```

HAL Configuration

```
GPIO_InitTypeDef GPIO_InitStruct;
// configure port A pin 10 as input
GPIO_InitStruct.Pin = GPIO_PIN_10; // declared in HAL gpio.c as (1<<10)
/* if there is another pin that should have the same settings (say
pin 11), we could do GPIO_PIN_10|GPIO_PIN_11 */
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_PULLUP;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
// configure port B pin 14 as output
GPIO_InitStruct.Pin = GPIO_PIN_14;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
```

Altimeter [Sensor] Reading

An altimeter provides the altitude above sea level in the range [-50 m, 150 m) as a signed 12-bit value, in the form of 2 separate bytes with the format A11A10A9A8xxxx (MSB) and A7A6A5A4A3A2A1A0 (LSB). If the sensor gives MSB = 0xAB and LSB = 0xCD, what is the altitude in metres? Note: x's in the format denotes reserved bits / don't cares – the values from these bits should be ignored.

```
#include <stdio.h>

void altitude_read(char*);

void print_altitude() {

    char altitude_val[2];
    altitude_read(altitude_val);

    int altitude_val_concat = ( altitude_val[0]<<4 & 0xFFFFFF00 ) |
                              ( altitude_val[1] & 0xFF );
   /* when a signed char/byte is promoted to a signed int, the MSB of
   the byte is duplicated into bits 31:8 (sign extension). These become
   bits 31:12 after <<4. & 0xFFFFF00 preserves these bits. */

    printf("Concatenated value (Hex)    : %x\n", altitude_val_concat);
    printf("Concatenated value (Decimal): %d\n", altitude_val_concat);

    float altitude_m = 50 + (float)altitude_val_concat * 200/4096;

    printf("The altitude is %f metres", altitude_m);

}

void altitude_read(char* altitude_val_raw)
{
    /* In reality, the data will be read from the altimeter through an
    interface such as I2C or SPI. We are hard-coding here for testing */
    altitude_val_raw[0] = 0xAB; //MSB
    altitude_val_raw[1] = 0xCD; //LSB
}
```

GPIO References

Mode(Input, Output, AF)

8.4.1 GPIO port mode register (GPIOx_MODER) (x =A to I)

Address offset:0x00

Reset value:

- 0xABFF FFFF (for port A)
- 0xFFFF FEBF (for port B)
- 0xFFFF FFFF (for ports C..G), I
- 0x0000 000F (for port H)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODE15[1:0]		MODE14[1:0]		MODE13[1:0]		MODE12[1:0]		MODE11[1:0]		MODE10[1:0]		MODE9[1:0]		MODE8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODE7[1:0]		MODE6[1:0]		MODE5[1:0]		MODE4[1:0]		MODE3[1:0]		MODE2[1:0]		MODE1[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **MODE[15:0][1:0]:** Port x configuration I/O pin y (y = 15 to 0)
These bits are written by software to configure the I/O mode.
00: Input mode
01: General purpose output mode
10: Alternate function mode
11: Analog mode (reset state)

Output Type (Push pull, Open-Drai)n

8.4.2 GPIO port output type register (GPIOx_OTYPER) (x = A to I)

Address offset: 0x04

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **OT[15:0]:** Port x configuration I/O pin y (y = 15 to 0)
These bits are written by software to configure the I/O output type.
0: Output push-pull (reset state)
1: Output open-drain

Pull-Up or Pull-Down

8.4.4 GPIO port pull-up/pull-down register (GPIOx_PUPDR) (x = A to I)

Address offset: 0x0C

Reset value: 0x6400 0000 (for port A)

Reset value: 0x0000 0100 (for port B)

Reset value: 0x0000 0000 (for other ports)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPD15[1:0]		PUPD14[1:0]		PUPD13[1:0]		PUPD12[1:0]		PUPD11[1:0]		PUPD10[1:0]		PUPD9[1:0]		PUPD8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPD7[1:0]		PUPD6[1:0]		PUPD5[1:0]		PUPD4[1:0]		PUPD3[1:0]		PUPD2[1:0]		PUPD1[1:0]		PUPD0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **PUPD[15:0][1:0]:** Port x configuration I/O pin y (y = 15 to 0)
These bits are written by software to configure the I/O pull-up or pull-down
00: No pull-up, pull-down
01: Pull-up
10: Pull-down
11: Reserved

Input Data

8.4.5 GPIO port input data register (GPIOx_IDR) (x = A to I)

Address offset: 0x10

Reset value: 0x0000 XXXX

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID15	ID14	ID13	ID12	ID11	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ID[15:0]:** Port x input data I/O pin y (y = 15 to 0)
These bits are read-only. They contain the input value of the corresponding I/O port.

Output Data

8.4.6 GPIO port output data register (GPIOx_ODR) (x = A to I)

Address offset: 0x14

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OD15	OD14	OD13	OD12	OD11	OD10	OD9	OD8	OD7	OD6	OD5	OD4	OD3	OD2	OD1	OD0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **OD[15:0]:** Port output data I/O pin y (y = 15 to 0)
These bits can be read and written by software.
Note: For atomic bit set/reset, the OD bits can be individually set and/or reset by writing to the GPIOx_BSRR or GPIOx_BRR registers (x = A..F).

Bit Set Reset(Output HIGH / HIGH & LOW at same time)

8.4.7 GPIO port bit set/reset register (GPIOx_BSRR) (x = A to I)

Address offset: 0x18

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Bits 31:16 **BR[15:0]:** Port x reset I/O pin y (y = 15 to 0)
These bits are write-only. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODx bit
1: Resets the corresponding ODx bit
Note: If both BSx and BRx are set, BSx has priority.

Bits 15:0 **BS[15:0]:** Port x set I/O pin y (y = 15 to 0)
These bits are write-only. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODx bit
1: Sets the corresponding ODx bit

Alternate Function Low[7:0] and High[15:8]

8.4.9 GPIO alternate function low register (GPIOx_AFRL) (x = A to I)

Address offset: 0x20

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFSEL7[3:0]				AFSEL6[3:0]				AFSEL5[3:0]				AFSEL4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFSEL3[3:0]				AFSEL2[3:0]				AFSEL1[3:0]				AFSEL0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

8.4.10 GPIO alternate function high register (GPIOx_AFRH) (x = A to I)

Address offset: 0x24

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFSEL15[3:0]				AFSEL14[3:0]				AFSEL13[3:0]				AFSEL12[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFSEL11[3:0]				AFSEL10[3:0]				AFSEL9[3:0]				AFSEL8[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **AFSEL[15:8][3:0]:** Alternate function selection for port x I/O pin y (y = 15 to 8)
These bits are written by software to configure alternate function I/Os.
0000: AF0
0001: AF1
0010: AF2
0011: AF3
0100: AF4
0101: AF5
0110: AF6
0111: AF7
1000: AF8
1001: AF9
1010: AF10
1011: AF11
1100: AF12
1101: AF13
1110: AF14
1111: AF15

*For both GPIOx_AFRH and GPIOx_AFRL

Bit Reset Register(For output LOW)

8.4.11 GPIO port bit reset register (GPIOx_BRR) (x = A to I)

Address offset: 0x28
Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **BR[15:0]:** Port x reset IO pin y (y = 15 to 0)
These bits are write-only. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODx bit
1: Reset the corresponding ODx bit

I2C Protocol

- Bus-based, multi-master, low-bandwidth, short distance, synchronous, serial, in-band addressing (7-bit slave address), 2 wire, half-duplex.
- Number of ICs connected is limited by bus capacitance.

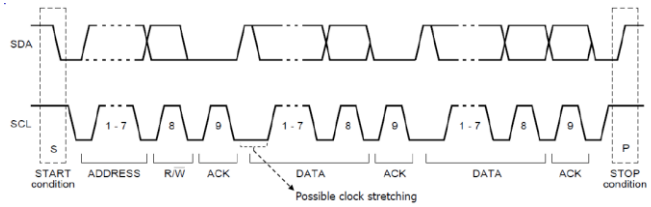
Speeds

Standard Mode	100 kbit/s	Fast-mode Plus	1 Mbit/s
Fast-mode	400 kbits/s	High-speed mode	3.4 Mbit/s

*Not all devices capable of all modes, standard mode most common.

I2C Signals

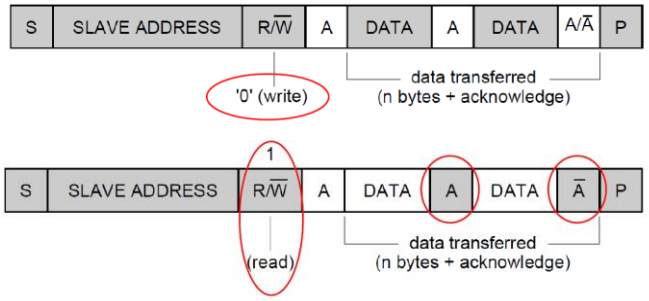
Open drain lines with pull up resistors.

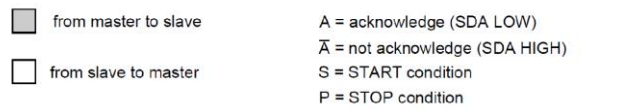


START(S): SDA goes low during SCL HIGH
STOP(P): SDA goes high during SCL HIGH
Rest of the data only changes when SCL is LOW

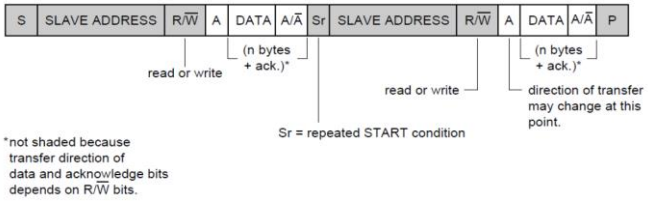
$$R/\overline{W}: 0 \rightarrow WRITE, 1 \rightarrow READ\ request$$

Master Transmitter/Receiver





- Note that for Master Transmitter, the master can either send a P or a Sr (repeated start) for change of direction. This prevents other Masters from controlling the bus.



- Master $\bar{A}$ (NACK) sent to tell slave that no more bytes required. Slave $\bar{A}$ has no standard meaning.

Clock Stretching

By the slave device, when it cannot receive or transmit another complete byte of data until it has performed some other function. It holds the clock line (SCL) LOW to force the master into the wait state.

Bit-banging

Other pins can also be used for a software-based (software-intensive, inefficient) implementation of I2C - a technique known as bit-banging.

Note

Library functions are written to favor ease of use / flexibility, but compromises on performance /battery life.

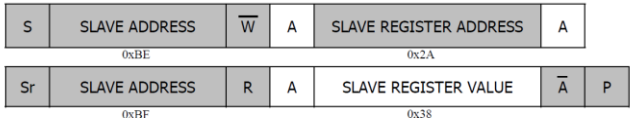
- Performance and battery life matter in many real-world applications when we move past the prototyping stage.
- Use of interrupts (instead of polling TXIS and RXNE) can make it much more efficient.

Write/Read of I2C Slave Register

Write

Read

Write then read normally using Sr



*Note that the addresses are different for this example. 0x5F is the 7bit slave address. 0xBE is the 8bit slave address + W[0] while 0xBF is the 8bit slave address + R[1]

Autoincrement

Can be used to read multiple registers separated by one bit in a single transaction.
The master reads two bytes in a single read transaction. The first byte is the content of the register whose address was written in the preceding write. The second byte is the content of the register with address++, i.e., a pointer internal to the slave keeps track of which register is read, and increments it with each byte read.

STM32 I2C Registers

I2C_CR1(Basic Ctrl/Enable)

Address offset: 0x00, Default value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	PECEN	ALERT EN	SMBD EN	SMBH EN	GCEN	WUPE N	NOSTR ETCH	SBC
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RXDMA EN	TXDMA EN	Res.	ANF OFF	DNF				ERRIE	TCIE	STOP IE	NACK IE	ADDR IE	RXIE	TXIE	PE
rw	rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

1. Write 0 to PE to disable and perform soft reset of interface
2. Set Analog and Digital noise Filters
3. Configure I2C Master Clock (SCLH and SCLL bits in I2C_ TIMINGR)
4. Enable interface by writing 1 to PE.

I2C_RXDR(Receive Data Register)

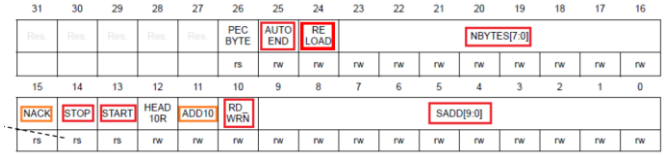
Bits 7:0 – Content from SIPO shift register which samples serial data on SDA in receive mode.

I2C_TXDR(Transmit Data Register)

Bits 7:0 – Data written here is copied into PISO shift register which sends the data serially on SDA in transmit mode.

I2C_CR2(Communication Control)

Information related to one transfer in master mode.



NBYTES used to indicate the amount of data received.

- If AUTOEND is set, then after NBYTES are received, a P is automatically generated. Else, TC* is set, which stretches SCL. STOP/START must be written to send P/Sr.
- If RELOAD is set, transfer is not completed after NBYTES are sent, and it must be reloaded.

START = 1 – Master mode and set S condition if bus is free

STOP = 1 – Generate P after current byte transfer

RD_WRN = 0 in write, 1 in read

I2C_ISR(Interrupt and Status Registers)

Status and interrupt status bits set by hardware and cleared due to action by software.

TCR: Transfer Complete Reload

- RELOAD = 1, NBYTES data transferred

TC: Transfer Complete (Master Mode)

- RELOAD = 0, AUTOEND = 0, NBYTES data transferred
- Cleared when sw sets START or STOP bit.

RXNE: Receive data register not empty.

- Received data is ready to be read from I2C_RXDR

Samuel Yow AY23/24 S1

- Remains set between 8th and 9th SCL Pulse before sending ACK.

TXIS: Transmit Interrupt Status

- Set when I2C_TXDR register is empty, after 9th SCL pulse when ACK received and NBYTES left > 0.

STOPF: Set when stop condition has appeared on bus.

NACKF: Set when byte sent is not acknowledged by receiver(slave)

Compute the total number of (a) Bytes sent (b) Bytes received (c) START bits sent (d) STOP bits sent (e) Acknowledges sent (f) Not-Acknowledges sent by the STM32 chip while executing this function

- `SENSOR_IO_ReadMultiple(DeviceAddr, (HTS221_T0_DEGC_X8 | 0x80), buffer, 2);`
  - S > 0xBE > A > 0x32|0x80 > A > Sr > 0xBF > A > 0x?? (T0_degC_x8 content) > A > 0x?? (T1_degC_x8 content) > A' > P
  - (a) 3 (b) 2 (c) 2 (d) 1 (e) 1 (f) 1
- `tmp = SENSOR_IO_Read(DeviceAddr, HTS221_T0_T1_DEGC_H2);`
  - S > 0xBE > A > 0x35 > A > Sr > 0xBF > A > 0x?? (T1/T0 msb content) > A' > P
  - (a) 3 (b) 1 (c) 2 (d) 1 (e) 0 (f) 1
- `SENSOR_IO_ReadMultiple(DeviceAddr, (HTS221_T0_OUT_L | 0x80), buffer, 4);`
  - S > 0xBE > A > 0x3C|0x80 > A > Sr > 0xBF > A > 0x?? (T0_OUT_L content) > A > 0x?? (T0_OUT_M content) > A > 0x?? (T1_OUT_L content) > A > 0x?? (T1_OUT_M content) > A' > P
  - (a) 3 (b) 4 (c) 2 (d) 1 (e) 3 (f) 1
- `SENSOR_IO_ReadMultiple(DeviceAddr, (HTS221_TEMP_OUT_L_REG | 0x80), buffer, 2);`
  - S > 0xBE > A > 0x2A|0x80 > A > Sr > 0xBF > A > 0x?? (TEMP_OUT_L content) > A > 0x?? (TEMP_OUT_H content) > A' > P
  - (a) 3 (b) 2 (c) 2 (d) 1 (e) 1 (f) 1
- Total : (a) 12 (b) 9 (c) 8 (d) 4 (e) 5 (f) 4
- Homework: Assume sending 1 byte takes approximately 10 bit periods (8 bits + A/A' + possible start/stop/overheads, etc.). How long does this function take if the transmission is in standard mode?
  - Ans: 2.1 ms

Note: In the solution above, 'Bytes' include slave address. The first byte has a specific meaning w.r.t I²C protocol and may or may not be included in the count, depending on who is counting and the context

5. Assuming the following register contents, compute the temperature reading returned by the function

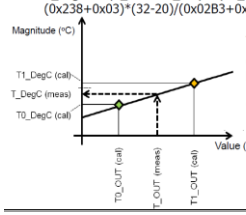
T0_degC_x8	0xA5
T1_degC_x8	0x07
T1/T0 msb	0xC4
T0_OUT_L	0xFD
T0_OUT_H	0xFF
T1_OUT_L	0xB3
T1_OUT_H	0x02
TEMP_OUT_L	0x38
TEMP_OUT_H	0x02

T1/T0 msb = 0xC4 = 0b11000100

T0_degC_x8_u16 = T1/T0 msb[1:0]<<8 | T0_degC_x8[7:0] = 0b00|0xA5 = 0xA5 = 0b0010100101

T1_degC_x8_u16 = T1/T0 msb[3:2]<<8 | T1_degC_x8[7:0] = 0b01|0x07 = 0x107 = 0b0100000111

- T0_degC = T0_degC_x8_u16 >> 3 = 0b**0010100**101 >> 3 = 0x14 = 20°C [floor(20.625)[4]]
- T1_degC = T1_degC_x8_u16 >> 3 = 0b**0100000**111 >> 3 = 0x20 = 32°C [floor(32.875)[4]]
- T0_out : 0xFFFD (-0x03, as the values are signed)
- T1_out : 0x02B3
- T_out : 0x238
- T_degC = (T_out - T0_out) * (T1_degC - T0_degC) / (T1_out - T0_out) + T0_degC = (0x238+0x03)*(32-20)/(0x02B3+0x03)+20 = **29.87°C** // all values should be cast to float.



T_degC is obtained through a linear interpolation

Given that two points (x_1, y_1) and (x_2, y_2) lie on a line, the equation of the line is

$$y = (x - x_1) \times \frac{y_2 - y_1}{x_2 - x_1} + y_1$$

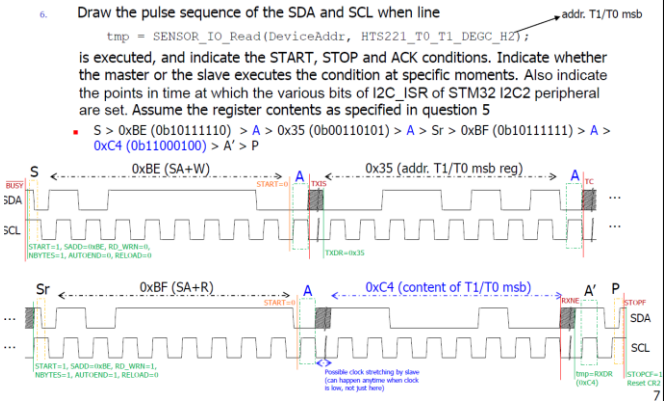
[4] >> 3 discards the 3 bits after the radix point, which reduces the precision as opposed to a floating-point division by 8. T_degC is (0x238+0x03)*(32.875-20.625) / (0x02B3+0x03)+20.625 = **30.70°C** if the exact values were used, i.e., if (float)T_degC_x8_u16/8 were used instead of T_degC_x8_u16>>3! (=0.1)

6. Draw the pulse sequence of the SDA and SCL when line

`tmp = SENSOR_IO_Read(DeviceAddr, HTS221_T0_T1_DEGC_H2);`

is executed, and indicate the START, STOP and ACK conditions. Indicate whether the master or the slave executes the condition at specific moments. Also indicate the points in time at which the various bits of I2C_ISR of STM32 I2C2 peripheral are set. Assume the register contents as specified in question 5

- S > 0xBE (0b10111110) > A > 0x35 (0b00110101) > A > Sr > 0xBF (0b10111111) > A > 0xC4 (0b11000100) > A' > P



Flags that are set

Interrupts & NVIC

Interrupts are more efficient since CPU can do other work. However, they are harder to design, debug and test.

There is **overhead/latency** as CPU must stop working, determine the device requesting the service, save its work (Status: R0-R3, SPI, LR…) before serving the request.

Interrupt Groups

System Exception

- Exception number 1-15
- Reset, NMI, Bus fault, Usage fault.

External Interrupt

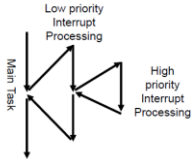
- Generated by on-chip peripherals

Priority Levels

Higher Priority(Lower levels) can pre-empt lower ones

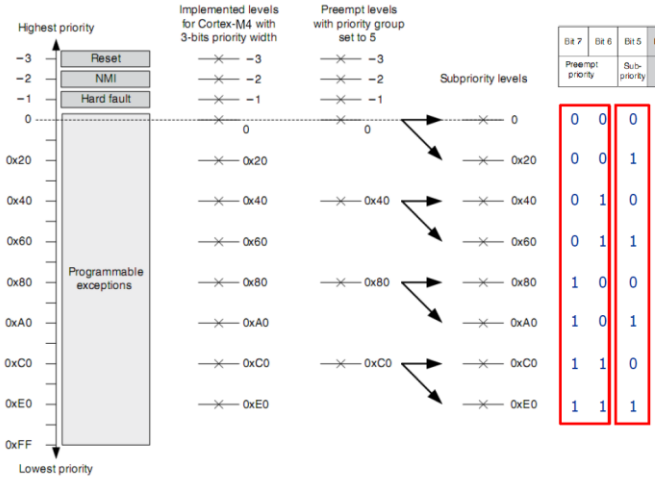
interrupt can pre-empt a lower priority one

- This is called nested interrupts
- 3 exceptions have fixed priority levels
  - Reset (-3); NMI (-2) and hard fault (-1)
  - Negative levels is to assure the highest priorities



*Note that exception number is used as a tie-breaker (sub-sub-priority) if 2 interrupts have the same priority setting.

There are up to **256** priority levels(*0xFF*), up to **128** levels of pre-emption.



a. IPR width = 2 bits; Priority Group 6

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3
Pre	Sub	Not Implemented (write		

Preempt Prio	Sub-prio
0x00	0x00
0x00	0x40
0x80	0x80
0x80	0xC0

b. IPR width = 2 bits; Priority Group 3

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3
Pre		Not Implemented (wri		

Preempt Prio	Sub-prio
0x00	0x00
0x40	0x40
0x80	0x80
0xC0	0xC0

c. IPR width = 2 bits; Priority Group 1

Bit 7	Bit 6	Bit 5	Bit 4	Bi
Pre		Not Implemented		

Preempt Prio	Sub-prio
0x00	0x00
0x40	0x40
0x80	0x80
0xC0	0xC0

d. IPR width = 2 bits; Priority Group 7

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3
Sub		Not Implemented (w		

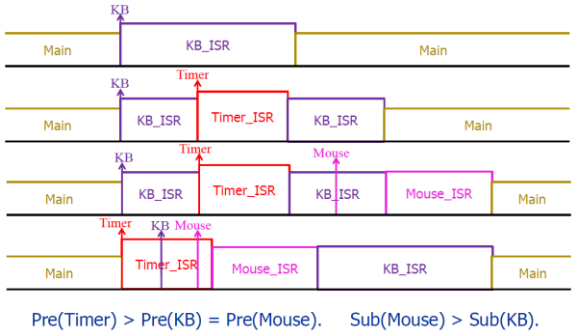
Preempt Prio	Sub-prio
0x00	0x00
0x00	0x40
0x00	0x80
0x00	0xC0

Pre-empt Priority

Defines whether an exc./int. handler can execute when CPU is running another exc./int. handler.

Sub Priority

When 2 exceptions/interrupts with the same pre-empt priority have occurred and are pending. Higher sub priority one will be handled first.



IPR Width

Each exc./int. has an Interrupt Priority Register (IPR) stores that stores its pre-empt(left bits) and sub-priority (right bits) values.

It has 8 bits but not all might be implemented. **For STM32, 4 bits, so 16 programmable priority levels**

- The split of implemented (available) bits between preempt and subpriority is determined by PRIGROUP (slide 15)

```
stm321475xx.h:
#define __NVIC_PRIO_BITS 4
// This is not a 'setting' and should not be modified
```

Priority Level Register with 3 and 4 bits implemented

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented			Not implemented				

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented			Not implemented				

The division of pre-empt and sub-priority bits is decided by the **PRIGROUP** in AICR (Application Interrupt and Reset Control Register) in System Control Block (SCB)

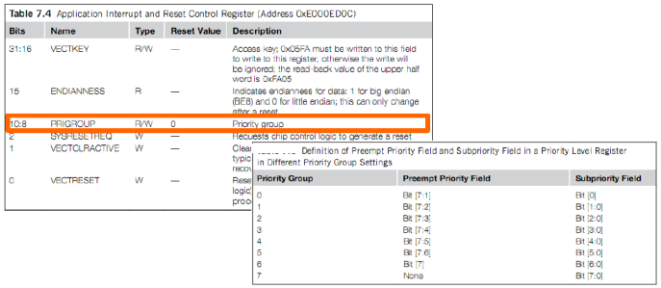


Table 7.4 Application Interrupt and Reset Control Register (Address 0xE000ED0C)

Bits	Name	Type	Reset Value	Description
31:16	VECTKEY	R/W	—	Access key: 0x05FA must be written to this field to write to this register, otherwise the write will be ignored. The read back value of the upper half word is 0xFA05
15	ENDIANNESS	R	—	Indicates endianness for data: 1 for big endian (BE8) and 0 for little endian; this can only change after a reset
10:8	PRIGROUP	R/W	0	Priority group
2	SYSRESETREQ	W	—	Requests chip control logic to generate a reset
1	VECTCLRACTIVE	W	—	Clear
0	VECTRESET	W	—	Reset

Definition of Preempt Priority Field and Subpriority Field in a Priority Level Register in Different Priority Group Settings

Priority Group	Preempt Priority Field	Subpriority Field
0	Bit [7:1]	Bit [0]
1	Bit [7:2]	Bit [1:0]
2	Bit [7:3]	Bit [2:0]
3	Bit [7:4]	Bit [3:0]
4	Bit [7:5]	Bit [4:0]
5	Bit [7:6]	Bit [5:0]
6	Bit [7]	Bit [6:0]
7	None	Bit [7:0]

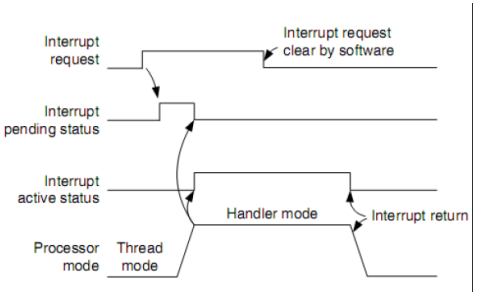
Interrupt Pending Behaviour

When the exc./int. is asserted, it is pended until the int./exc. handler starts. (exc./int. is waiting for the cpu to process)

When the int. handler is executed, it then becomes active and the pending bit is cleared.

Pending bit can be cleared before CPU response by writing to an interrupt control register in NVIC

*Note: ONLY happens when enabled.



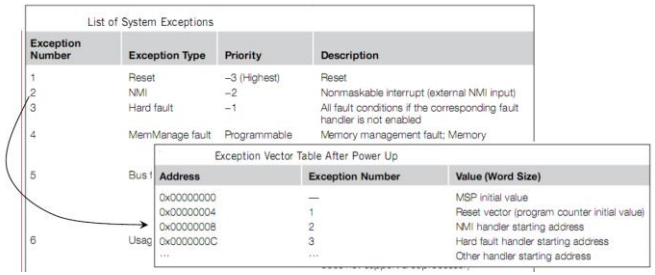
An int. that is signalled several times before it is processed will treat it as a single int. request.

If the int. source continues to hold the int. request signal active, the int. will be pended again at the end of the ISR. (When interrupt not cleared)

Vector Table

Part of memory used to store the starting addresses of the various exception/interrupt handlers.

- Address arranged to exc./int. number x 4 as an interrupt vector takes 4 bytes.
- Vector table starts at memory address 0 which is the stack pointer. The first exception is at 0x04.
- Exception 40 is found in 40x4 = 160 = 0x 0000 00A0
- Can be relocated to another memory location by setting Vector Table Offset Register (VTOR) in SCB



List of System Exceptions

Exception Number	Exception Type	Priority	Description
1	Reset	–3 (Highest)	Reset
2	NMI	–2	Nonmaskable interrupt (external NMI input)
3	Hard fault	–1	All fault conditions if the corresponding fault handler is not enabled
4	MemManage fault	Programmable	Memory management fault; Memory
5	Bus f		
6	Usag		

Exception Vector Table After Power Up

Address	Exception Number	Value (Word Size)
0x00000000	—	MSP initial value
0x00000004	1	Reset vector (program counter initial value)
0x00000008	2	NMI handler starting address
0x0000000C	3	Hard fault handler starting address
…	…	Other handler starting address

6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000 0058
7	14	settable	EXTI1	EXTI Line1 interrupt	0x0000 005C
8	15	settable	EXTI2	EXTI Line2 interrupt	0x0000 0060
9	16	settable	EXTI3	EXTI Line3 interrupt	0x0000 0064

23	30	settable	EXTI9_5	EXTI Line[9:5] interrupts	0x0000 009C
40	47	settable	EXTI15_10	EXTI Line[15:10] interrupts	0x0000 00E0

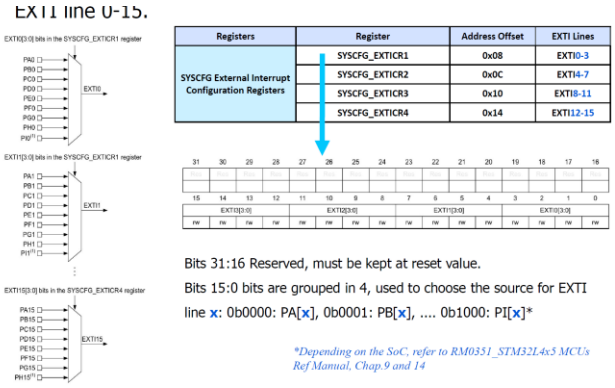
Note that EXTI0 – EXTI4 are treated as separate interrupts, while lines in EXTI15_10 and EXTI9_5 treated as the same interrupt (within each EXTI) and will have the same priority.

EXTI

Management of external and internal asynchronous events/interrupts and generates event request to NVIC.

SYSCFG_EXTRICRn

EXTI line 0-15:



Registers	Register	Address Offset	EXTI Lines
SYSCFG External Interrupt Configuration Registers	SYSCFG_EXTICR1	0x08	EXTI0-3
	SYSCFG_EXTICR2	0x0C	EXTI4-7
	SYSCFG_EXTICR3	0x10	EXTI8-11
	SYSCFG_EXTICR4	0x14	EXTI12-15

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 bits are grouped in 4, used to choose the source for EXTI line x: 0b0000: PA[x], 0b0001: PB[x], ... 0b1000: PI[x]*

**Depending on the SoC, refer to RM0351_STM32L4x5 MCUs Ref Manual, Chap.9 and 14*

Each CRx register can configure pin x. Then in that register, write different values to activate interrupt at different ports.
E.g, EXTICR1[15:12] is for pin 3.
Write 0000 to activate PA[3].
0001, 0010, 0011 & 0100 for PB[3], PC[3],PD[3] & PE[3].

EXTI Memory Map

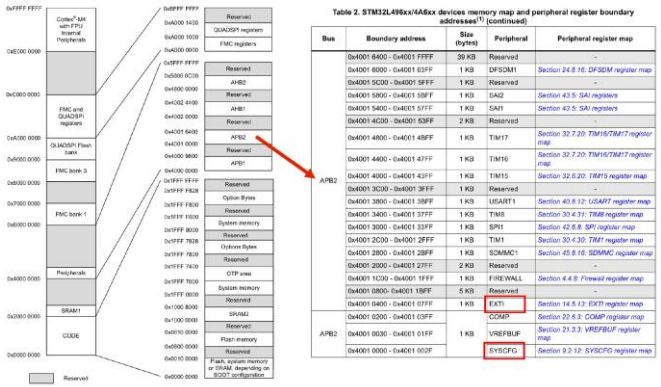


Table 2. STM32L496xx/4A6xx devices memory map and peripheral register boundary addresses[1] (continued)

Bus	Boundary address	Size (bytes)	Peripheral	Peripheral register map
APB2	0x4001 6400 - 0x4001 FFFF	39 KB	Reserved	-
	0x4001 6000 - 0x4001 63FF	1 KB	DFSDM1	Section 24.8.16: DFSDM register map
	0x4001 5C00 - 0x4001 5FFF	1 KB	Reserved	-
	0x4001 5800 - 0x4001 5BFF	1 KB	SAI2	Section 43.5: SAI registers
	0x4001 5400 - 0x4001 57FF	1 KB	SAI1	Section 43.5: SAI registers
	0x4001 4C00 - 0x4001 53FF	2 KB	Reserved	-
	0x4001 4800 - 0x4001 4BFF	1 KB	TIM17	Section 32.7.20: TIM15/TIM17 register map
	0x4001 4400 - 0x4001 47FF	1 KB	TIM16	Section 32.7.20: TIM15/TIM17 register map
	0x4001 4000 - 0x4001 43FF	1 KB	TIM15	Section 32.6.20: TIM15 register map
	0x4001 3C00 - 0x4001 3FFF	1 KB	Reserved	-
	0x4001 3800 - 0x4001 3BFF	1 KB	USART1	Section 40.8.12: USART register map
	0x4001 3400 - 0x4001 37FF	1 KB	TIM8	Section 30.4.31: TIM8 register map
	0x4001 3000 - 0x4001 33FF	1 KB	SPI1	Section 42.8.8: SPI register map
	0x4001 2C00 - 0x4001 2FFF	1 KB	TIM1	Section 30.4.30: TIM1 register map
	0x4001 2800 - 0x4001 2BFF	1 KB	SDMMC1	Section 45.8.16: SDMMC register map
	0x4001 2000 - 0x4001 27FF	2 KB	Reserved	-
	0x4001 1C00 - 0x4001 1FFF	1 KB	FIREWALL	Section 4.4.8: Firewall register map
	0x4001 0800 - 0x4001 1BFF	5 KB	Reserved	-
	0x4001 0400 - 0x4001 07FF	1 KB	EXTI	Section 14.5.13: EXTI register map
APB2	0x4001 0200 - 0x4001 03FF	1 KB	COMP	Section 22.6.3: COMP register map
	0x4001 0030 - 0x4001 01FF		VREFBUF	Section 21.3.3: VREFBUF register map
	0x4001 0000 - 0x4001 002F		SYSCFG	Section 9.2.12: SYSCFG register map

EXTI Registers

EXTI_IMR	Interrupt Mask, bits [15:0]. 1 to enable interrupts
EXTI_RTSR1	Rising Edge interrupt. 1 to enable
EXTI_FTSR1	Falling Edge interrupt. 1 to enable.
EXTI_PR1	Pending interrupt. 1 when selected trigger request has occurred. Must be cleared by writing 1 to the bit. Else it will keep entering the handler.

*Can set both EXTI_RTSR1 and EXTI_RTSR1 for the same interrupt lines

Steps
1.Enable AHB2 bus for GPIO Ports
2. Configure SYSCFG_EXTICRn to chose Port and Pin as EXTI Source.
3. Configure EXTI_IMRn
4. Configure EXTRI_FTSRn/EXTRI_RTSRn
5. Enable IRQ in NVIC

Interrupt Programming

Can have more than one, as each bit is one interrupt. For STM32, 82 external interrupts so there are 3 (82//32 = 3) of each register.

Interrupt Registers

NVIC_ISER(32 bits, RW): enable interrupts (write 1).
NVIC_ICER(32 bits, RW): disable interrupts (write 0).
*Both can be read to check interrupt status. (INT enabled or disabled)

NVIC_ISPR(32 bits RW): force interrupts into pending state
NVIC_ICPR(32 bits RW): clear pending interrupts
*Both NVIC_ISPR, NVIC_ICPR can be used to check for pending interrupts

NVIC_IPR: Write priority level (0-255) of interrupt.
NVIC_IABR: Determine which interrupts are active. Bit set to 1 if interrupt handler is being executed.

System Exceptions

Controls are part of System Control Block (SCB) a CORE PERIPHERAL.
SHPR: Priority of system exceptions with programmable priority levels
ICSR: To set and clear pending status of system exceptions and determine currently executing exception/interrupt number using VECTACTIVE.

SHCSR: Enable usage faults, memory management faults, bus faults. Pending and active status also available.
PRIMASK: set to 1, only allow NMI, hard fault and reset exceptions. PRI $\geq$ 0 are disabled.
FAULTMASK: Allow NMI, reset exceptions. PRI ≥ -1 disabled.
BASEPRI(8bit): For disabling exceptions/interrupts of priority number $\geq$ BASEPRI. 0: disables BASEPRI.

Steps

1. Set priority group setting: number of bits for pre-empt priorities. NOTE: less is better as system loses time saving and restoring system state during preemptions.
2. Set priority level of interrupt (Default 0).
3. Implement ISR
4. Configure interrupt generation, peripheral + mcu.
5. Clear pending status in NVIC
6. Enable interrupt in NVIC

Implementation

1. PUSH all registers modified + (R0 – R3, R12, LR, PC, xPSR).
2. Check cause for interrupt by checking which pin, or whether falling/rising edge.
3. Do the bare minimum to deal with the cause. Higher priority – make ISR shorter, no blocking code. `Volatile` flag for variables modified.
4. Clear pending status in interrupt device.
5. POP all registers pushed at beginning.

EXTI Registers

EXTI_IMR1

14.5.1 Interrupt mask register 1 (EXTI_IMR1)

Address offset: 0x00

Reset value: 0xFF82 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IM31	IM30	IM29	IM28	IM27	IM26	IM25	IM24	IM23	IM22	IM21	IM20	IM19	IM18	IM17	IM16
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IM15	IM14	IM13	IM12	IM11	IM10	IM9	IM8	IM7	IM6	IM5	IM4	IM3	IM2	IM1	IM0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **IMx:** Interrupt Mask on line x (x = 31 to 0)
0: Interrupt request from Line x is masked
1: Interrupt request from Line x is not masked

Note: The reset value for the direct lines (line 17, lines from 23 to 34, line 39) is set to '1' in order to enable the interrupt by default.

EXTI_RTSR1

14.5.3 Rising trigger selection register 1 (EXTI_RTSR1)

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	RT22	RT21	RT20	RT19	RT18	Res.	RT16
									rw	rw	rw	rw	rw		rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RT15	RT14	RT13	RT12	RT11	RT10	RT9	RT8	RT7	RT6	RT5	RT4	RT3	RT2	RT1	RT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:23 Reserved, must be kept at reset value.

Bits 22:18 **RTx:** Rising trigger event configuration bit of line x (x = 22 to 18)
0: Rising trigger disabled (for Event and Interrupt) for input line
1: Rising trigger enabled (for Event and Interrupt) for input line

Bit 17 Reserved, must be kept at reset value.

Bits 16:0 **RTx:** Rising trigger event configuration bit of line x (x = 16 to 0)
0: Rising trigger disabled (for Event and Interrupt) for input line
1: Rising trigger enabled (for Event and Interrupt) for input line

Note: The configurable wakeup lines are edge-triggered. No glitch must be generated on these lines. If a rising edge on a configurable interrupt line occurs during a write operation in the EXTI_RTSR register, the pending bit is not set.

Rising and falling edge triggers can be set for the same interrupt line. In this case, both generate a trigger condition.

EXTI_FTSR1

14.5.4 Falling trigger selection register 1 (EXTI_FTSR1)

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	FT22	FT21	FT20	FT19	FT18	Res.	FT16
									rw	rw	rw	rw	rw		rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FT15	FT14	FT13	FT12	FT11	FT10	FT9	FT8	FT7	FT6	FT5	FT4	FT3	FT2	FT1	FT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:23 Reserved, must be kept at reset value.

Bits 22:18 **FTx:** Falling trigger event configuration bit of line x (x = 22 to 18)
0: Falling trigger disabled (for Event and Interrupt) for input line
1: Falling trigger enabled (for Event and Interrupt) for input line

Bit 17 Reserved, must be kept at reset value.

Bits 16:0 **FTx:** Falling trigger event configuration bit of line x (x = 16 to 0)
0: Falling trigger disabled (for Event and Interrupt) for input line
1: Falling trigger enabled (for Event and Interrupt) for input line

Note: The configurable wakeup lines are edge-triggered. No glitch must be generated on these lines. If a falling edge on a configurable interrupt line occurs during a write operation to the EXTI_FTSR register, the pending bit is not set.

Rising and falling edge triggers can be set for the same interrupt line. In this case, both generate a trigger condition.

EXTI_PR1

14.5.6 Pending register 1 (EXTI_PR1)

Address offset: 0x14

Reset value: undefined

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	PIF22	PIF21	PIF20	PIF19	PIF18	Res.	PIF16
									rc_w1	rc_w1	rc_w1	rc_w1	rc_w1		rc_w1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PIF15	PIF14	PIF13	PIF12	PIF11	PIF10	PIF9	PIF8	PIF7	PIF6	PIF5	PIF4	PIF3	PIF2	PIF1	PIF0
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

Bits 31:23 Reserved, must be kept at reset value.

Bits 22:18 **PIFx:** Pending interrupt flag on line x (x = 22 to 18)
0: No trigger request occurred
1: Selected trigger request occurred
This bit is set when the selected edge event arrives on the interrupt line. This bit is cleared by writing a '1' to the bit.

Bit 17 Reserved, must be kept at reset value.

Bits 16:0 **PIFx:** Pending interrupt flag on line x (x = 16 to 0)
0: No trigger request occurred
1: Selected trigger request occurred
This bit is set when the selected edge event arrives on the interrupt line. This bit is cleared by writing a '1' to the bit.

Interrupt Programming

```
void __enable_irq(); // Clear PRIMASK
void __disable_irq(); // Set PRIMASK
void __set_PRIMASK(uint32_t priMask); // Set PRIMASK to value priMask
uint32_t __get_PRIMASK(void);// Read the PRIMASK value
void __set_FAULTMASK(uint32_t faultMask); //Set or clear FAULTMASK
uint32_t __get_FAULTMASK(void); // Read the FAULTMASK value
void __set_BASEPRI(uint32_t basePri); // Set the base priority register
uint32_t __get_BASEPRI(void); // Return the content of the base priority register
__set_BASEPRI(0x60);    // Disable interrupts with priorities from 0x60-0xFF
```

CMSIS functions	Description
`void NVIC_EnableIRQ(IRQn_Type IRQn)`	Enables an interrupt or exception.
`void NVIC_DisableIRQ(IRQn_Type IRQn)`	Disables an interrupt or exception.
`void NVIC_SetPendingIRQ(IRQn_Type IRQn)`	Sets the pending status of interrupt or exception to 1.
`void NVIC_ClearPendingIRQ(IRQn_Type IRQn)`	Clears the pending status of interrupt or exception to 0.
`uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)`	Reads the pending status of interrupt or exception. This function returns non-zero value if the pending status is set to 1.
`uint32_t NVIC_GetActive(IRQn_Type IRQn);`	Read the active bit for an external interrupt
`void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)`	Sets the priority of an interrupt or exception with configurable priority level to 1.
`uint32_t NVIC_GetPriority(IRQn_Type IRQn)`	Reads the priority of an interrupt or exception with configurable priority level. This function return the current priority level.
`void NVIC_SetPriorityGrouping(uint32_t PriorityGroup);`	Set the Priority Grouping in NVIC Interrupt Controller
`uint32_t NVIC_EncodePriority (uint32_t PriorityGroup, uint32_t PreemptPriority, uint32_t SubPriority);`	Encode the priority for an interrupt. The returned priority value can be used for NVIC_SetPriority(...) function

The input parameter *IRQn* is the IRQ number

Example (Demo)

EXTI configuration (generation of Interrupts) was covered in Chapter

- PriorityGroup=5
- PreemptPriority=0b11
- SubPriority=0b00
- Full IPR contents:
- 0b11000000 = 0xC0

SetPriority() expects right-aligned priority, but IPR is left-aligned

```
NVIC_SetPriorityGrouping(5);
NVIC_SetPriority(EXTI15_10_IRQn, 0x0C);
// Set priority level to 0xC0 = 1100 0000
NVIC_ClearPendingIRQ(EXTI15_10_IRQn);
NVIC_EnableIRQ(EXTI15_10_IRQn);
```

Table 7.5 Definition of Preempt Priority Field and Subpriority Field in a Priority Level Register in Different Priority Group Settings

Priority Group	Preempt Priority Field	Subpriority Field
0	Bit [7:1]	Bit [0]
1	Bit [7:2]	Bit [1:0]
2	Bit [7:3]	Bit [2:0]
3	Bit [7:4]	Bit [3:0]
4	Bit [7:5]	Bit [4:0]
5	Bit [7:6]	Bit [5:0]
6	Bit [7]	Bit [6:0]
7	None	Bit [7:0]

```
#include "stm321475xx.h " //contains definition of __NVIC_PRIO_BITS, xxxIRQns etc.
int main(void) {
        uint32_t ans, PG=5, PP=0b11, SP=0b00;
        NVIC_SetPriorityGrouping(5);
        ans = NVIC_EncodePriority(PG,PP,SP); // ans = 0x0C
        NVIC_SetPriority(EXTI15_10_IRQn,ans);
/* Priority in the IPR for IRQn = 40 set to 0xC0 (SetPriority shifts it up for us)
NVIC->IP[((uint32_t)IRQn)] = (uint8_t)((priority << (8U - __NVIC_PRIO_BITS)) & (uint32_t)0xFFUL);
*/
}
```

stm32l4xx_it.c

```
void EXTI15_10_IRQHandler(void)
{
/* For vector table to be set up correctly,
the interrupt handler name should match the
interrupt handler name in
startup_stm321475vgtx.s */
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_10);
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_11);
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_14);
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_15);
}
```

```
void HAL_GPIO_EXTI_IRQHandler(uint16_t
GPIO_Pin)
{
  /* EXTI line interrupt detected */
  if(__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) !=
                                    0x00u)
  {
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
    HAL_GPIO_EXTI_Callback(GPIO_Pin);
  }
}
```

```
__weak void HAL_GPIO_EXTI_Callback(uint16_t
                                   GPIO_Pin)
{
/* Prevent unused argument(s) compilation
warning */
  (void)(GPIO_Pin);

  /* NOTE: This function should not be
modified, when the callback is needed, the
HAL_GPIO_EXTI_Callback could be implemented
in the user file (main.c) */
}
```

main.c

```
// Implemented by the user in main.c
// this one overrides the _weak function
with the same name
void HAL_GPIO_EXTI_Callback(uint16_t
GPIO_Pin)
{
  // Actual handler implementation here
  // following the guidelines in slide 11-13
}
```

Which register(s) do the following functions modify?

(i) void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority);
(ii) void NVIC_SetPriorityGrouping(uint32_t PriorityGroup);
(iii) uint32_t NVIC_EncodePriority (uint32_t PriorityGroup, uint32_t PreemptPriority, uint32_t SubPriority);

(i) Writes to the appropriate SHPR for system exceptions / appropriate IPR for external interrupts.
(ii) Writes into PRIGROUP field (bits 10:8) of the Application Interrupt and Reset Control Register.
(iii) This function is just to encode the priority in the form required by the function NVIC_SetPriority(), and does not write directly to any register.

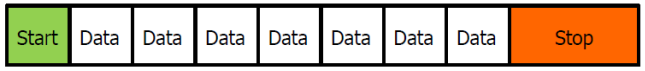
UART Intro

*Note that I2C (100000)and UART (115200)are quite comparable in speed.

Hardware for long-distance, asynchronous, serial, point-to-point, peer-peer, full duplex communication.

Transmitter: Takes bytes of data and transmits bits sequentially using a PISO shift register.

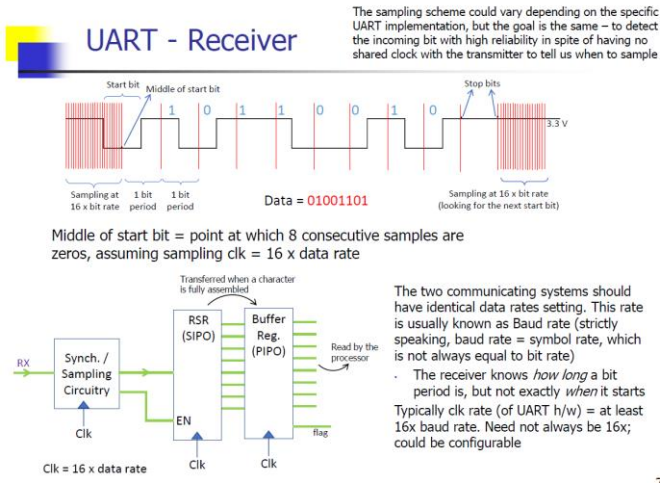
Receiver: Assembly bits into bytes using SIPO.



Each character (8 bits) sent as a frame with

- Start: Logic LOW
- Data bits (word length, usually 8) LSB first normally
- Optional parity bit(bit error detection)
  - Parity bit is because UART is meant for long distance and there is a high chance for data corruption. Common check is using XOR
- End: One or more logic HIGH bits
- Line remains high until next start bit.

Receiver



- Has more complicated hardware. Has a clock signal that samples 16x the data rate. It samples the data to look for the incoming start bit.
- Start bit is only valid if it lasts for at least one-half of the bit time, else it is discarded.
- Data can be copied into a buffer register so that a new character can be received while the current character is being read.
- FIFO buffer memory can be inserted between receiver shift register and host processor for time to read data and prevents loss of data at high rates.

Transmitter

When data is deposited into shift register, UART hardware generates start bit, shifts data bits onto line, generates parity bit and generates stop bit. During this time, the UART maintains a busy flag so a new character is not deposited. Note that there is TX and RX, 2 different circuits (shift registers inc) for transmitted and received characters.

- STM32L475 has three universal synchronous/asynchronous receiver transmitters (USART1, USART2 and USART3), two universal asynchronous receiver transmitters (UART4, UART5), as well as a low-power UART (LPUART)
- The exact features depends on which of the above 3 is being used. USARTx is the most versatile, supporting almost all standard UART features (except a FIFO – only transmit and receive buffer registers are available). These include
  - Configurable number of stop bits (0.5-2)
  - 7-9 bit data (word) length
  - Parity generation and checking (odd, even)
  - Autobaud rate detection etc.
- In addition, it also supports some features which are typically not supported by UARTs, such as
  - Multiprocessor communication with addressing
  - Synchronous mode etc.

UART HAL Functions & Registers

- Control Registers : Configured by `HAL_UART_Init()`
  - Control register 1 (USART_CR1) : Parity settings, word length (inclusive of parity bit, if parity is used), transmit and receive enables, interrupt enables
  - Control register 2 (USART_CR2) : Number of stop bits, advanced features
  - Control register 3 (USART_CR3) : Sampling mode, error interrupt enable
  - Baud rate register (USART_BRR) : Baud rate setting
- Data Registers
  - Receive data register (USART_RDR) : CPU reads this to get the received data
  - Transmit data register (USART_TDR) : CPU writes data to be transmitted into it
- Status Register
  - Interrupt and Status Register (USART_ISR). Has status bits/flags such as
    - BUSY: Busy flag
    - TXE: Transmit data register empty
    - RXNE: Read data register not empty
    - Error flags such as overrun error (new data received when previous data in USART_RDR is still unread by the CPU), parity error, framing error (stop bit not logic 1), etc.
  - `HAL_UART_Transmit()` writes to USART_TDR if TXE bit in USART_ISR is set
  - `HAL_UART_Receive()` reads from USART_RDR if RXNE bit in USART_ISR is set

4. [Self-study] Two STM32 chips communicate via UART. You need to send one 32-bit intege from STM32_A, which should be printed to the debug console by STM32_B. How can this be accomplished?

   UART allows only characters (7-9 bits in case of STM32). However, any data can be sen via UART if STM32_A splits the data into bytes and sends them over, and STM32_B receives them and assembles them back appropriately. Two possible ways in which this can be done are shown below.

```
//method 1: using a pointer gimmick - computationally simpler, but the
endianness of the transmitter and receiver should be the same.

      // in STM32_A
          int a = 0x41424344; // the integer to be sent;
          char *buf_a = (char*)&a;
          HAL_UART_Transmit(&huart, buf_a, 4, Timeout);

      // in STM32_B
          char buf_b[4];
          HAL_UART_Receive(&huart, buf_b, 4, Timeout);
          int b = *((int*)buf_b);
          printf("%d\n", b);

// method 2: using shift-and-or: similar in spirit to how we dealt with reading
separate MSB and LSB values from a sensor.

      // in STM32_A
          int c = 0x41424344; // the integer to be sent;
          char buf_c[4];
          buf_c[0] = c & 0xFF;
          buf_c[1] = c>>8 & 0xFF;
          buf_c[2] = c>>16 & 0xFF;
          buf_c[3] = c>>24 & 0xFF;
          HAL_UART_Transmit(&huart, buf_c, 4, Timeout);

      // in STM32_B
          char buf_d[4];
          HAL_UART_Receive(&huart, buf_d, 4, Timeout);
          int d = buf_d[3]<<24|buf_d[2]<<16|buf_d[1]<<8|buf_d[0];
          printf("%d\n", d);
```