Mtrx2700

RISC - Reduced Instruction Set

Buses between components (STM32), 8-bit bus size:

- I-bus — *instruction* bus, used for fetching program instructions
- D-bus — *data* bus, used for r/w data
- S-bus — *system* bus, used for I/O, GPIO and peripherals

ALU - Arithmetic-Logic Unit

Assembly code

```
.byte 1, 2, 0 @ make an array of bytes
```

Assembly ⟶ machine code

Structure

```
.data
  // variables & constants
  ascii_string: .asciz "MTRX2700!\0" @ Define a null-terminated string

.text
  // instructions
```

Data sizes

.byte	1 byte
half word	2 bytes
.word	4 bytes

Main Arithmetic Ops

For all operations Op2 is optional, and can be left out so its acts on the existing data. i.e., ADD Rd, Rn $\implies$ Rd = Rn + Rd ≡ Rd += Rn

Opcode	Operands	Func
ADC	Rd, Rn, Op2	$Rd = Rn + Op2 + C$
ADD	Rd, Rn, Op2	$Rd = Rn + Op2$
MOV	Rd, Rn	$Rd = Rn$
SUB	Rd, Rn, Op2	$Rd = Rn - Op2$
MUL	Rd, Rn, Rm	$Rd = Rn \times Rm^{\dagger}$
SDIV*	Rd, Rn, Op2	$Rd = Rn \div Op2^{\ddagger}$

† clips to least significant 32 bits

‡ cannot set flags (i.e., cannot use postfix S),

* UDIV is the unsigned alternative.

Main Logic Ops

Opcode	Operands	Func
AND	Rd, Rn, Op2	$Rd = Rn \wedge Op2$
BIC - bit clear	Rd, Rn, Op2	$Rd = Rn \wedge \neg\ Op2$
EOR	Rd, Rn, Op2	$Rd = Rn \oplus Op2$
ORR	Rd, Rn, Op2	$Rd = Rn \vee Op2$
CMP	Rd, Rn	Compare Rd and Rn, setting flags

Flags

Add S to opcodes to make them set the conditional flags, i.e., SUBS which sets said flags

Flag	Condition
N	res < 0
Z	res $= 0$
C	Has a carry bit
V	Leads to overflow

Then after said operation can conditionally execute operations based on the flags,

Suffix Code	Condition	Meaing
EQ	$Z = 1$	Equal
NE	$Z = 0$	Not equal
GE	$N = V$	$\geq$
LT	$N \neq V$	$<$
GT	$Z = 0 \wedge N = V$	$>$
LE	$Z = 1 \wedge N \neq V$	$\leq$

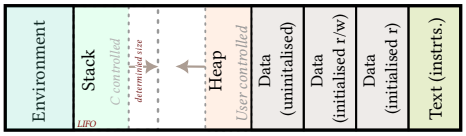
All the 'condition' stuff is just what the CPU does, it is usually used after a CMP instruction.

```
CMP R0, R1; // compare R0 & R1, setting flags

// IT block scoping the GT stuff
ITT GT;
CMPGT R2, R3; // compare R2 & R3, setting flags
MOVGT R4, R5; // if R2 > R3, do R4 = R5
```

Use B to branch, and the BL to branch and store in the LR (link register), and then can get back with BX LR ≡ MOV PC, LR.

Memory



Stack

Used for functions with stack frames - where variables & local parameters are pushed to it. LIFO memory (last in-first out)

Heap

Dynamically sized memory, largely unstructured where data can be stored anywhere a user desires. Used for 'object' storage.

The SP is a descending pointer, so adding elements **decreases** it while removing/popping elements **increases** it.

Registers

- 32-bits (4-bytes) wide

R0-R7 (8 registers) are low, operands registers.

R8-R12 (8 registers) are high registers.

R13 — SP, stack pointer

R14 — LR, link register

R15 — PC, program counter

Memory Ops

```
LDR R0, =0x1234 @ loads the hex value (0x1234) into Register 0

LDR R1, [R0] @ loads Register 1 with the VALUE of Register 0
LDR R3, R4 @ load Register 3 with the ADDRESS of Register 4
```

Array-based loading:

```
.data
  my_arr: .byte 0, 1, 2, 3

.text
  LDR R0, =my_arr @ load addr of my_arr into Reigster 0
  LDRB R1, [R0, #1] @ load `1` into R1

  LDRB R1, [R0, #1]! @ pre-increment, so load while incrementing R0
  LDRB R1, [R0], #2 @ post-increment, load then increment R0 by 2
```

Use qualifiers on LDR like B to load just a byte (8 bits), or H to load a half-word (2 bytes). By default loads a word (4 bytes).

LDR takes a **value** *from* **memory**. While STR puts a register **value** *into* **memory**.

```
@ storing data
STR R3, [R4]; @ store the data in R3, at the address pointed to by R4
```

Stack Operations

Can PUSH and POP from the stack,

```
PUSH {R1} @ push the value on R1 to the stack
POP {R1} @ takes whatever is in the stack and dumps it in R1, in this case old R1 val

STR R0, [SP, #10] @ store R0 value at particular spot on stack
POP {R0-R3} @ mutli register popping
```

The stack pointer, SP, points to where in memory the stack is being read from. The *stack* is a **LIFO** operation, so the as elements are pushed in they are taken out first.

Types of memory:

1. Cache, fastest - L1, L2
2. Primary - SRAM, DRAM
3. Secondary - SDD, HDD

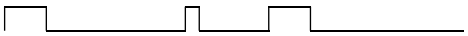
Flash storage is persistent and used for firmware; EEPROM non-volatile storage for more config based data — as harder to r/w.

GPIO

General Purpose I/O

Better to use interrupts to ensure that code is not stuck polling, and can better react to changes without burning CPU cycles. Ensure that interrupts (IRQ) don't take too long, and prevent any other code from executing.

Interrupt flow:



system is 'nudged' out of program flow to interrupt IRQ code

Can use in C with,

```
void enable_interrupt() {
  __disable_irq(); // good to disable while editing to stop heebie geebies

  RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN; // enable system config controller
  SYSCFG->EXTICR[0] = SYSCFG_EXTICR1_EXTI0_PA; // enable for PA0
  EXTI->RTSR |= EXTI_RTSR_TR0; // set to rising edge of EXT1 line 0
  EXTI->IMR |= EXTI_IMR_MR0; // enable interrupt

  // tell modules it is enabled
  NVIC_SetPriority(EXTI0_IRQn, 1);
  NVIC_EnableIRQ(EXTI0_IRQn);

  __enable_irq();
}

// the actual interrupt handler
void EXTI0_IRQHandler(void) {
  if (on_button_press != 0x00) {
    on_button_press();
  }

  EXTI->PR |= EXTI_PR_PR0; // reset the interrupt (so it doesn't keep firing until the next trigger)
}
```

I²C

I²C (i-squared c) allows for cross module communication, across a shared bus using two shared wires.

1. SCL - serial clock
2. SDA - serial data

It is a synchronous form, where controllers address the peripherals to send commands.

U(S)ART

Universal Asynchronous Receiver-Transmitter, where a shared clock doesn't need to used between devices.

- Data speed: bits per second (bps)
- Baud: rate at which the signal changes

$\texttt{baud} = \frac{f_{CK}}{\texttt{USART Div}}$

Use USART_ISR to check for the status of the USART, and it has to be cleared sometimes on error.

```
.syntax unified
.thumb

.global main
.thumb_func

.type main, %function
#include "initialise.s"

.data
@ define variables
```

```
.align
incoming_buffer: .space 255

incoming_counter: .byte 255 @ size of buffer

@ Define a string
tx_string: .asciz "Test Test\r\n"
tx_length: .byte 27


.text

main:
  BL initialise_power
  @BL change_clock_speed
  BL enable_peripheral_clocks
  BL enable_uart

  B tx_loop

  @ To read in data, we need to use a memory buffer to store the incoming bytes
  @ Get pointers to the buffer and counter memory areas
  LDR R6, =incoming_buffer
  LDR R7, =incoming_counter

  @ dereference the memory for the maximum buffer size, store it in R7
  LDRB R7, [R7]

  @ Keep a pointer that counts how many bytes have been received
  LDR R8, =0x00

@ continue reading forever (NOTE: eventually it will run out of memory)
read_loop:
  LDR R0, =UART @ the base address for the register to set up UART
  LDR R1, [R0, USART_ISR] @ load the status of the UART

  TST R1, 1 << UART_ORE | 1 << UART_FE  @ 'AND' the current status with the bit mask that we are interested in

  BNE clear_error

  TST R1, 1 << UART_RXNE @ 'AND' the current status with the bit mask that we are interested in

  BEQ read_loop @ loop back to check status again if the flag indicates there is no byte waiting

  LDRB R3, [R0, USART_RDR] @ load the lowest byte (RDR bits [0:7] for an 8 bit read)
  STRB R3, [R6, R8]
  ADD R8, #1

  CMP R7, R8
  BGT no_reset
  LDR R8, =0x00

no_reset:

  @ load the status of the UART
  LDR R1, [R0, USART_RQR]
  ORR R1, 1 << UART_RXFRQ
  STR R1, [R0, USART_RQR]

  BGT read_loop

clear_error:
  @ Clear the overrun/frame error flag in the register USART_ICR (see page 897)
  LDR R1, [R0, USART_ICR]
  ORR R1, 1 << UART_ORECF | 1 << UART_FECF @ clear the flags (by setting flags to 1)
  STR R1, [R0, USART_ICR]
  B read_loop


tx_loop:
  @ the base address for the register to set up UART
  LDR R0, =UART

  @ load the memory addresses of the buffer and length information
  LDR R3, =tx_string
  LDR R4, =tx_length

  @ dereference the length variable
  @ notice how memory address R4 is replaced by the value that was at that memory address
  LDR R4, [R4]

tx_uart:
  LDR R1, [R0, USART_ISR] @ load the status of the UART
  ANDS R1, 1 << UART_TXE  @ 'AND' the current status with the bit mask that we are interested in
                @ NOTE, the ANDS is used so that if the result is '0' the z register flag is set

  @ loop back to check status again if the flag indicates there is no byte waiting
  BEQ tx_uart

  @ load the next value in the string into the transmit buffer for the specified UART
  LDRB R5, [R3], #1
  STRB R5, [R0, USART_TDR]

  CMP R5, #0 @ look for the end of the string
  BEQ delay_loop

  @ keep looping while there are more characters to send
  B tx_uart

delay_loop:
  LDR R9, =0x7fff

delay_inner:
  SUBS R9, #1
  BGT delay_inner
  B tx_loop @ go back to start
```

1. Need to enable the pins, can be done through the HAL
2. Turn it on & setup GPIO; `GPIOx_OSPEEDR`, `UARTxEN`, `RCC_APB1ENR`
3. Set UART params:
   i. Baud rate, `USART_BRR`
   ii. Frame Construction, `USART_CR1` - word length & parity
   iii. Error handling
4. Read & write to the serial port

SPI

Used for much faster, specific peripheral code communication. Has `SCK`, `MISO` (Master In-Slave Out), `MOSI` & `SS` pins.

STM32 Peripherals

Timers

Timers are much better than a basic hardware clock,

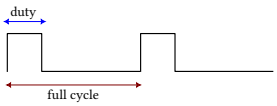
```
waste_time:
  LDR R0, =0xffffffff   @ load large counter
.loop:                  @ label for the loop func
  SUBS R0, R0, #1       @ subtract and set flags
  BNE .loop             @ loop until zero
  BX LR                 @ return
```

Need to use `RCC` (Reset & Clock Control) for handling timer based operations.

1. Enable in `RCC` with `RCC_APB1ENR`
2. Enable the timer itself with `TIMx_CR1`
   - Can use a prescaler (`TIMx_PSC`) upon the clock to make it count slower by a factor ρ i.e., Base clock 8MHz, then a $p = 8000 \implies f = \frac{8}{8000} = 1$ kHz
3. Read timer value with `TIMx_CNT`
   - A 32-bit counter overflows over 2^{32} counts, with $\rho = 1$ @ 536.87s
   - To make the period shorter use `TIMx_ARR` to change the max value @ overflow point usually `0xFFFF FFFF`

PWM Control

Using Pulse Width Modulation control, can turn on and off power rapidly for different lengths of the duty cycle.



ADC/DAC

Analogue-Digital Convertors, need to set the V_{ref} so they work out what is the maximum 'defined' voltage. resolution $= \frac{V_{\text{ref}}}{2^n}$

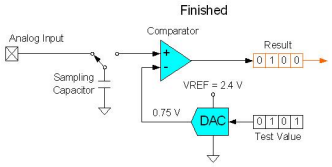
Read from `ADC1->DR` which is a 12 bit register, so needs to be adjusted to the particular range $-0 - 3.3V$,

```
uint16_t ReadADC(uint32_t channel) {
  ADC1->SQR1 = 0;
  ADC1->SQR1 |= channel << ADC_SQR1_SQ1_Pos; // set the request for channel specified
  // request the process to start
  ADC1->CR |= ADC_CR_ADSTART;
  // Wait for the end of conversion
  while((ADC1->ISR &= ADC_ISR_EOS) != ADC_ISR_EOS);
  return ADC1->DR;
}
```

SAR

Successive approximation, flow:

1. Uses a series of capacitors to *sample & hold* the input voltage V_{ref}
2. An op-amp compares V_{ref} to the result of a DAC, V_{DAC}
3. The DAC is continually 'improved' through the results of the comparator using a successive-approximation register changes the MSB first to LSB
4. DAC is read to get the digital approximation



HAL

Hardware abstraction layer, used/controlled through `STM32CubeMX` MX, which allows for automatic code generation and configuration of peripherals.

`#include "stm32f303xc.h"` — HAL code

Misc

Block checker checksum: `BCC` $= b_1 \oplus b_2 \oplus b_3 \oplus \ldots \oplus b_n$

Bit Manipulation

To target a specific bit and perform one of these operations:

CLEAR	inv mask `AND`	`1010 & ~0010 = 1000`
SET	mask `OR`	`1010 | 0001 = 1011`
TOGGLE	inv mask `AND`	`1010 ^ 0010 = 1000`

Game design in *Escape Rooms* — Cubescape

- Event-driven programming
- TCP/IP based network stack, async http libraries ↑
- Use own *'English-based'* scripting language, with inbuilt redundancy

RTOS

Real-Time Operations Systems, these allow for precise control over timing & deterministic behaviour.

Need to deal with resource management, where you may need `MUTEXs` to ensure that one thread/process has access to one resource at a time.

Also handle the switching of context, which is how a normal computer can process multiple times at the same time.

1. **Hard** real time, system needs real-time or in total failure.
2. **Soft** real time, system will have degraded performance.

Watchdog programs can check whether the system is responsive, and reset them if they are not.

STM32 with C

ANSI C standard defines how C is made, so it is a portable language which can interrupt ASM instructions inline.

Better to use `#include <stdint.h>` to ensure that integer types have consistent sizes, like `uint32_t` .

Need to also use `volatile` to ensure that code in `C` is not optimised away. There are different optimisation levels which can strip out code.

1. Source File
2. Preprocessor
3. Compiler
4. Assembler
5. Linker

Header guards,

```
// used to prevent header code from being included multiple times
#ifndef __MAIN_H
#define __MAIN_H
  // the header code & definitions - public interface
#endif /* __MAIN_H */
```

AAPCS

Arm Architecture Procedure Call Standard, rules dictating how to pass data, use memory & registers.

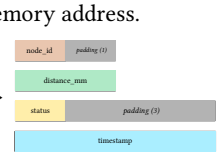
`R0-R1`	Argument/result
`R2-R3`	Argument/scratch register (used for intermediary calcs)
`R4-R8`	Variables — so have to be `PUSH`ed and `POP`ed to ensure data integrity across calls, also need to save the `SP`

As registers are 32-bit (4 byte), types that overflow use pair registers on an even bound i.e., `R0-R1`. If the data is too large or cannot be determined, the result is stored in a register passed in as an argument to the call.

Byte alignment, according to *GCC* rules, each type is aligned to its own size. So a `int`, `long` is 4 bytes (`uint32_t`) and **aligned** to those 4 bytes. So when storing this data it must be at a multiple of 4 starting memory address.

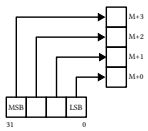
```
typedef struct {
  uint8_t  node_id;
  uint16_t distance_mm;
  uint8_t  status;
  uint32_t timestamp;
} SensorNode;
```

→



** trick to best compact a data `struct`, start from largest data object to smallest.*

The `STM32F3Discovery` is a LE (little endian board), so stored by LSB:



Communication

Need to serialise data structures to send them over the wire to be able allow to send properly, can send it as a string.

```
// Seralise data
snprintf(buf, buf_len, "%d,%d,%d,%.6f,%lu,%d",
        (int)data->rawX,
        (int)data->rawY,
        (int)data->rawZ,
        data->heading,
        (unsigned long)data->timestamp,
    (unsigned int)data->display_mode);

// Deseralise
int parsed = sscanf(snapshot, "%d,%d,%d,%f,%lu,%d",
&x, &y, &z, &heading, &ts, &display_mode);
    if (parsed != 6) return 0;

data->rawX = (int16_t)x;
data->rawY = (int16_t)y;
data->rawZ = (int16_t)z;
data->heading = (double)heading;
data->timestamp = (uint32_t)ts;
data->display_mode = (uint8_t)display_mode;
```

There are specified data formats like XML, JSON and pickle which have defined standards for sending messages.

Structs

Used for *structured* storage of data,

```
typedef struct {
  uint32_t BIT0: 1; // bit fields (0 or 1)
  uint32_t BIT1: 1;
  uint32_t BYTE2: 8; // max val: 2^8 - 1 = 255
} RegisterBits;
```

Hex to Binary

Hex	Binary	Decimal	0x7	0111	7
0x0	0000	0	0x8	1000	8
0x1	0001	1	0x9	1001	9
0x2	0010	2	0xA	1010	10
0x3	0011	3	0xB	1011	11
0x4	0100	4	0xC	1100	12
0x5	0101	5	0xD	1101	13
0x6	0110	6	0xE	1110	14
			0xF	1111	15

C reference

Functions

Functions, user-defined - can take in args of certain type and return of other types.

Function prototype: `char my_func(int my_arg, float* p, char my_str[])`. With `char` being the return type. Defined at top of file or in header.

Main function,

```
int main(int argc, char** argv) {
  // argc number of args pass in through argv
  // argv[0] == name of program
  return 1; // 1 successful, 0 not
}
```

`sizeof` the size of a type in **bytes**, `sizeof((int) 10) = 4`

Pointers

```
int d = 10;
int *p = &d; // points to 10 value
int pd = *d; // val: 10
*p = 500; // now d is 500 too
```

`void*`, special pointer which points to data of *any* type in memory. Has to be casted to be used. i.e., `void *ptr = &b; // b is 10 (int *)ptr // 10`

`NULL`, special pointer which a *nothing* value

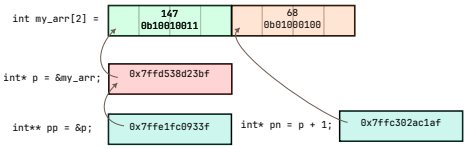
Pointer arithmetic

```
int my_arr[] = {0, 1, 2, 3, 4, 5};
int* my_ptr = my_arr; // points to 0, don't need &
as my_arr is already ptr
*(my_ptr + 2) // 2
*(++my_ptr) // 1
```

`const int* ptr`, cannot be used to change data.

		Change addr?	Change data?
`const int *ptr;` `int const *ptr;`	Pointer to constant	✓	X
`int * const ptr;`	Constant pointer	X	✓
`const int const *ptr;`	Constant pointer to constant	X	X

functions cannot return a pointer, as the pointed to memory would have been deallocated/freed



Scope

```
#include <stdio.h>
int G = 10; // file scope
static int S_FILE = 20;
void P(int a, int b); // Function Prototype Scope

void S_F() {
    static int S_FUNC = 0; S_FUNC++; // persists between calls
}

int main() {
    int L = 30; // Function Scope
    printf("FileG:%d, StaticF:%d, Extern:%d\n", G, S_FILE, EXT_VAR);
    { int L = 50; printf("Block L: %d\n", L); } // Nested/Block
    printf("Func L: %d\n", L); // Back to L=30
    P(100, 200);
    return 0;
}
// File Scope: Definition for EXT_VAR
int EXT_VAR = 99;

void P(int x, int y) { // func defined above
    printf("  P Result: %d\n", x + y);
}
```

Multifile Code

```
// main.c
#include "animal.h"

int main() {
  frog(); // now has access to frog func!
  BEST_ANIMAL == "🐥"; // 1/true
  WORST_ANIMAL == "🦏"; // 1/true
  SEAL == "🦭"; // runtime error
}
```

```
// animal.c
const char* frog() {
  return "🐸";
}
const char* SEAL = "🦭";
```

```
// animal.h
const char* frog();
#define BEST_ANIMAL "🐥"
const char* WORST_ANIMAL = "🦏";
```

Header guard, `#ifndef MY_HEADER_H`, `#define MY_HEADER_H` then `#endif` - stops header content being linked multiple times.