NTURACING

MCU Programming

W3: Peripherals

by 羅紀翔

2024-25 FSAE Season

Agenda

NTURACING

I/O Models	1
2	C Technique 3 Built-in Functions
Peripherals	3
4	LAB 3 UART I

I/O Models

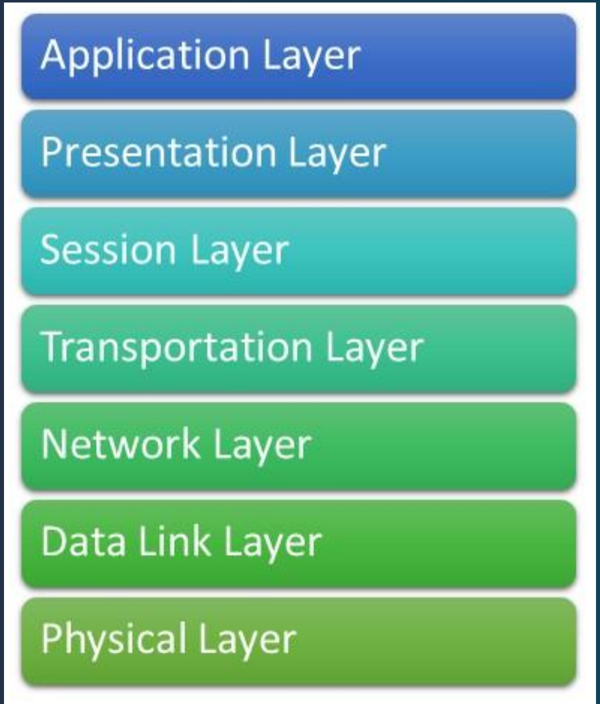
References: CH2-5, *The Definitive Guide*; 事件驅動伺服器：原理和實例 (https://hackmd.io/@sysprog/event-driven-server)

I/O Models

NTURACING

What is I/O

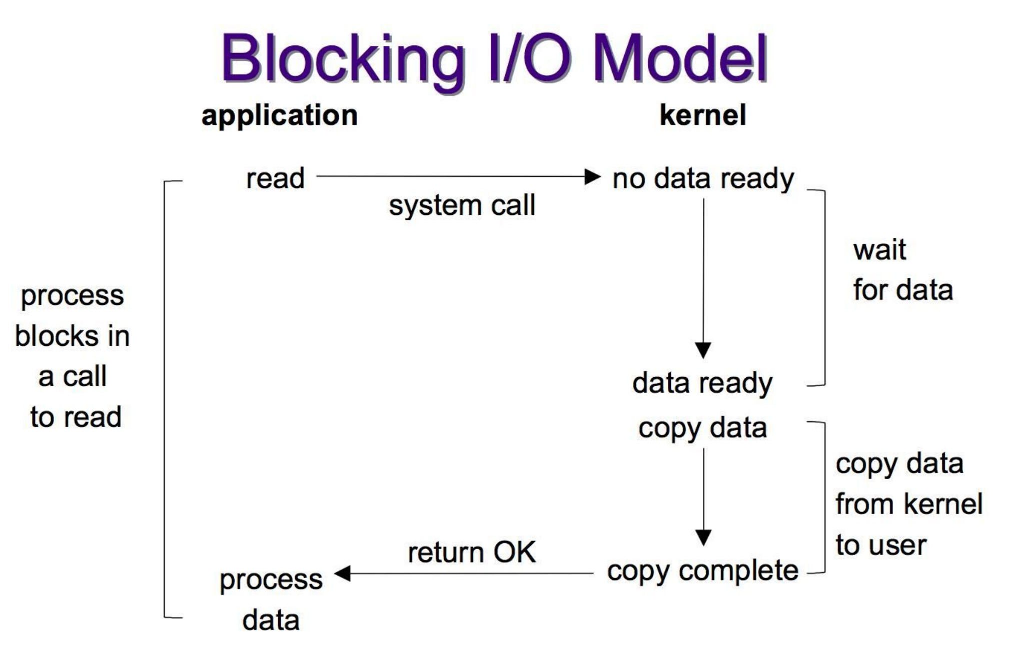
- Communication with other devices
- Various protocols: UART, I2C, SPI, CAN, USB, Ethernet, PCIe, etc.
- Open Systems Interconnection (OSI) 7-layer model
    - Physical: Wire, RF frequency/modulation
    - Data Link: Medium access control
    - Network: Route data among different nodes
    - Transport: Multiplex physical channels into logical ones
    - Application: What does the data mean
- Different protocols specify different requirements of different layers
    - CAN bus specify physical and data link layers
    - CANopen specify network and application layers



I/O Models

I/O in Program

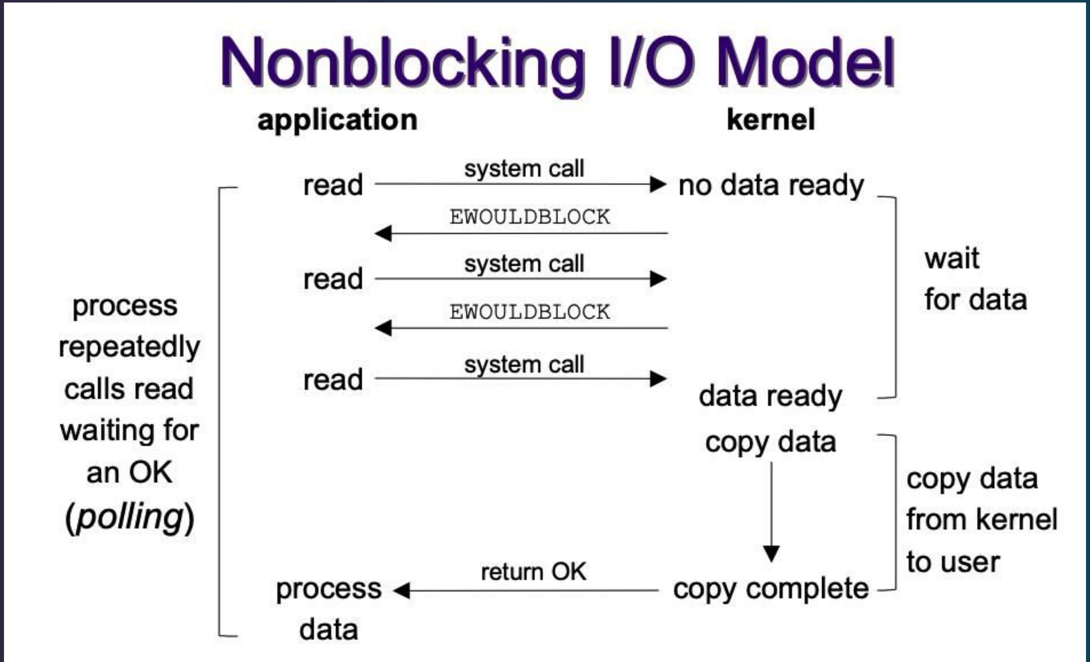
- C functions: printf(), scanf(), read(), write()
- Blocking and synchronous call: Does not requrn until I/O is complete
- However, I/Os are SLOW!!!
    - UART: 115200 bps, SPI: 10 Mbps, CAN: 1Mbps
- CPU: 170 MHz (STM32G4), every instruction can manipulate 4 bytes => 580 MBps!!!
- CPU does nothing while waiting for I/O => needs other way to better utilize CPU



I/O Models

Non-blocking I/O

- Use `O_NONBLOCK` flag in `fcntl` to make `read` non-blocking

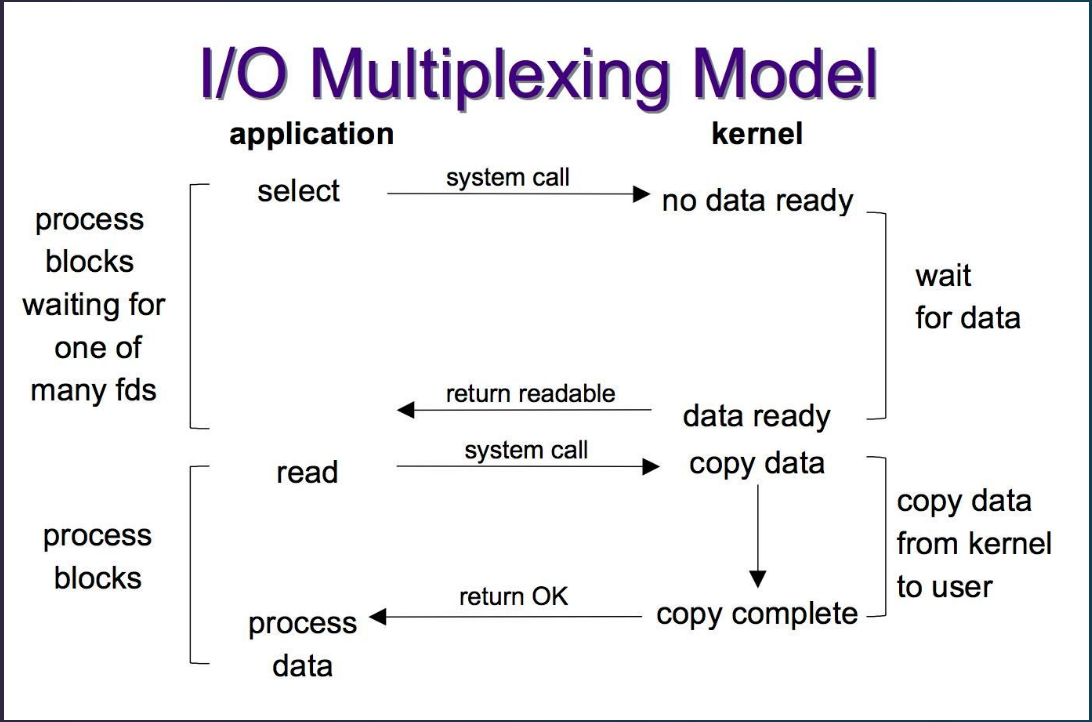


I/O Models

I/O Multiplexing

- `select`, `poll`, `epoll` to monitor multiple I/Os at a time

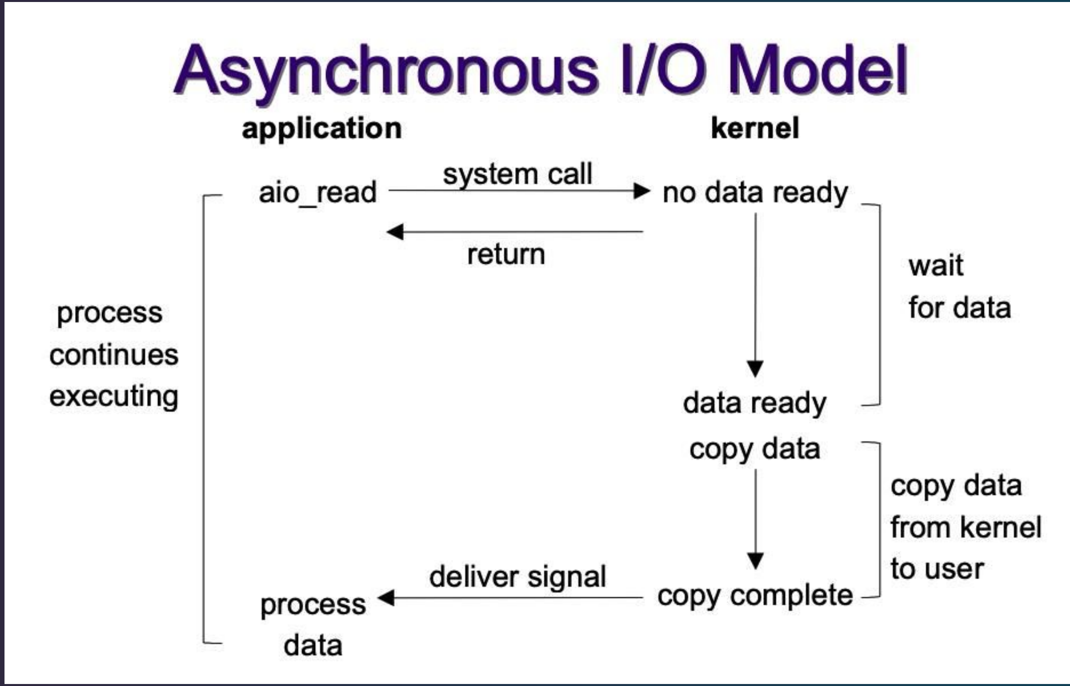
```
nfd = epoll_wait(epoll_fd, events,
MAX_EVENTS, -1);
for (i = 0; i < nfd; i++) {
    if (events[i].events & EPOLLIN) {
        read(events[i].data.fd, ...);
        ...
    }
}
```



I/O Models

I/O Multiplexing

- kernel waits and performs I/O in the background and notify application when done
- Notify via signal or callbacks

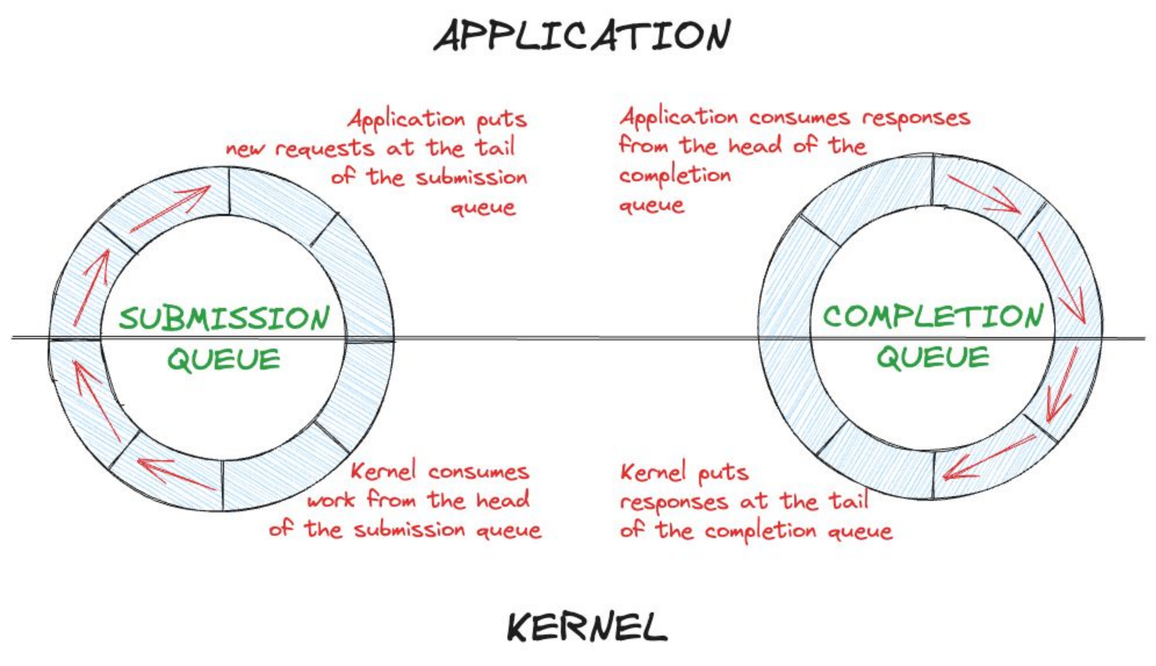


I/O Models

NTURACING

io_uring

- Asynchronous
- The performance king of I/O
- Application put requests onto submission queue
- Kernel processes requests from submission queue and put result onto completion queue
- Appliction check result from completion queue



I/O Models

Microcontroller

- STM32 HAL provides three I/O APIs:
    - Blocking
        - Polling (`HAL_UART_Transmit()`)
    - Asynchronous
        - Interrupt-driven (`HAL_UART_TransmitIT()`)
        - DMA (`HAL_UART_TransmitDMA()`)
- Most other vendors also provide the same three APIs
- Since there is no kernel that helps you, everything is doen via hardware or in the application code
- Embedded OSes (such as Zephyr) may provide other models discussed earlier

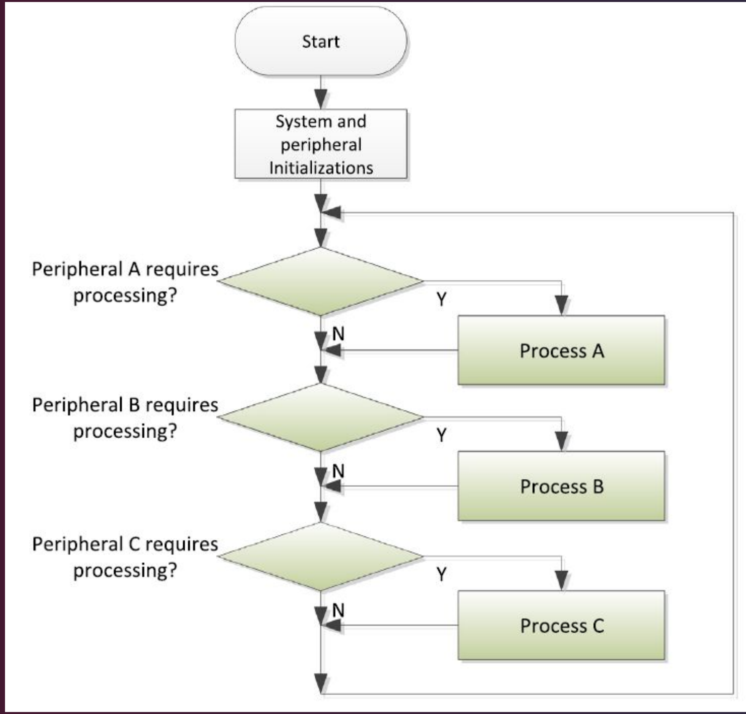
I/O Models

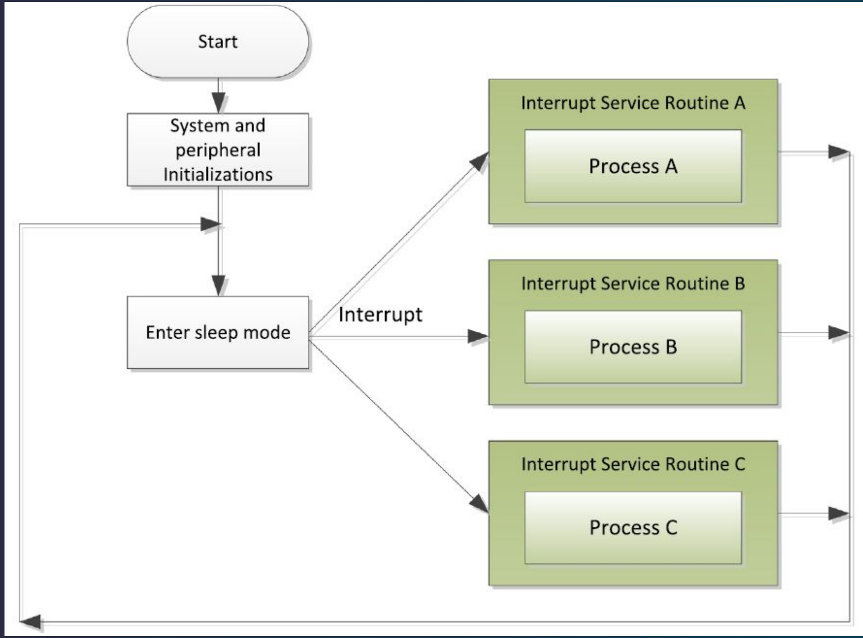
Which Model to Use?

- Blocking
  - Easy
- Non-blocking
  - More performant
  - More real-time
  - Acts as a slave

I/O Models

Software flow: polling or event-driven





C Technique 3: Built-in Functions

References: GCC reference: https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

Built-in

NTURACING

GCC Built-in Functions

- Functions provided by GCC
  - Bit manipulation
    - `__builtin_ffs()`: Returns one plus the index of the least significant 1-bit of x, or if x is zero, returns zero.
    - `__builtin_clz()`: Returns the number of leading 0-bits in x, starting at the most significant bit position. If x is 0, the result is undefined.
    - `__builtin_popcount()`: Returns the number of 1-bits in x.
  - Change endiness: `__builtin_bswap16()`, `__builtin_bswap32()`, ...
  - Check types are the same: `__builtin_types_compatible_p()`
  - Check for compile time constant: `__builtin_constant_p()`
- GCC built-in function: https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

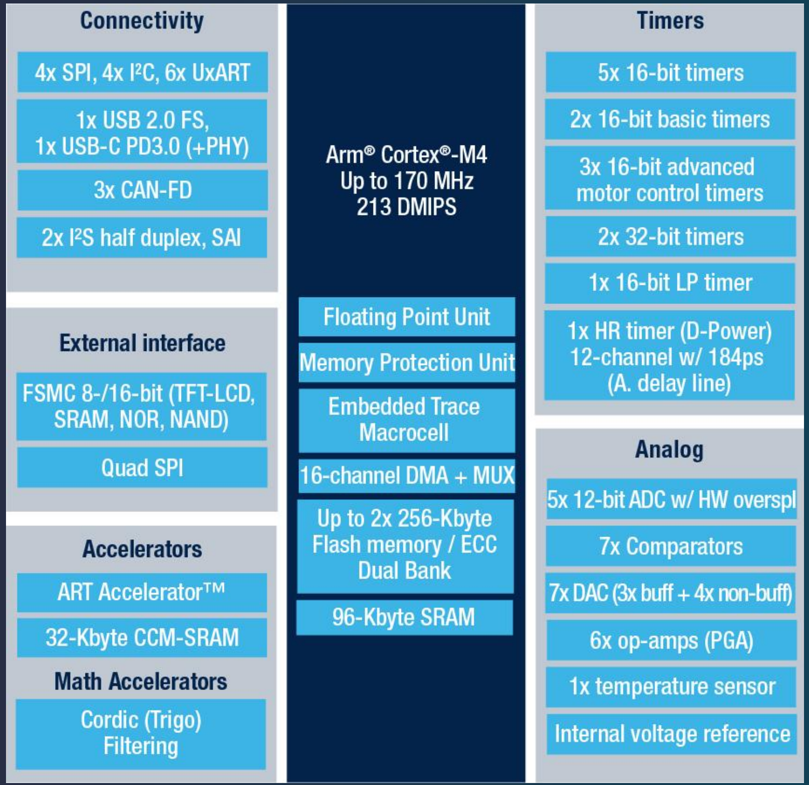
Peripherals

References: Ch 6-1, 8, 9, *Mastering STM32*;

Peripherals

What are Peripherals in Microcontrollers

- Hardware that connects microcontrollers to the world
- Implement communcation protocols (i.e. UART, I2C, etc) to lighten CPU's workload
    - CPU can implement a protocol by controlling the output of GPIOs in a way that conform the protocol, which is called "bit-banging "
- Timers, Analog connectivity, Accelerators

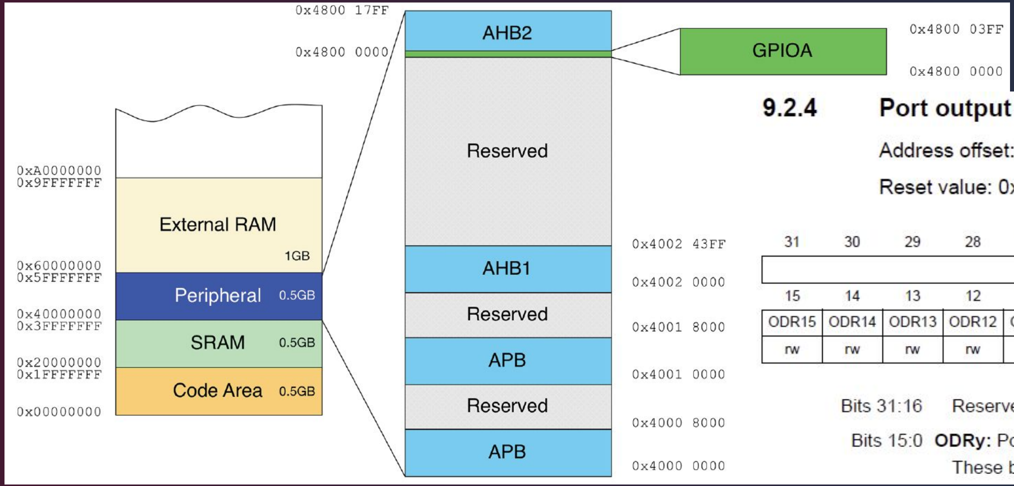


Peripherals

NTURACING

Memory-Mapped I/O (MMIO)

- CPU used to have special instructions that controls peripherals
- Modern CPU controls peripherals via memory
    - Peripheral "**registers** " are mapped to memory address
    - The "**side effects** " of the program



9.2.4 Port output data register (GPIOx_ODR) (x=A..G)

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
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 **ODRy:** Port output data (y= 0 .. 15)
These bits can be read and written by software and can be accessed in Word mode only.

Turn on the LED, the Hard Way

- LD2 connected to PA5, GPIOA at 0x4800 0000 for STM32G4

9.4.1 GPIO port mode register (GPIOx_MODER) (x =A to G)

Address offset:0x00

Reset value: 0xABFF FFFF (for port A)

Reset value: 0xFFFF FEBF (for port B)

Reset value: 0xFFFF FFFF (for ports C..G)

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)

9.4.6 GPIO port output data register (GPIOx_ODR) (x = A to G)

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 register (x = A..F).

Turn on the LED, the Hard Way (con'd)

```c
int main(void) {
  // Address of the GPIOA->MODER register
  volatile uint32_t *GPIOA_MODER = (uint32_t *)0x48000000;

  // Address of the GPIOA->ODR register
  volatile uint32_t *GPIOA_ODR = (uint32_t *)(0x48000000 + 0x14);

  // This ensures that the peripheral is enabled and connected to the AHB2 bus
  __HAL_RCC_GPIOA_CLK_ENABLE();

  *GPIOA_MODER = *GPIOA_MODER | 0x400;  // Sets MODER[11:10] = 0x1
  *GPIOA_ODR = *GPIOA_ODR | 0x20;  // Sets ODR[5] = 0x1, that is pulls PA5 high

  while (1)
    ;
}
```

Peripherals

UART in Polling Mode

```
HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size, uint32_t
Timeout);
HAL_StatusTypeDef HAL_UART_Receive(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size, uint32_t
Timeout);
```

- `huart`: The UART hardware handler (as an object-oriented design)
- `Timeout`: The maximum time to wait in ms before abotring the task
- Blocks the CPU while waiting for the transmission/reception to complete

Peripherals

Hardware Handler in HAL

```
typedef struct __UART_HandleTypeDef {
  USART_TypeDef                *Instance;        /*!< UART registers base address        */
  UART_InitTypeDef             Init;             /*!< UART communication parameters      */
  uint8_t                      *pTxBuffPtr;      /*!< Pointer to UART Tx transfer Buffer */
  uint16_t                     TxXferSize;       /*!< UART Tx Transfer size              */
  __IO uint16_t                TxXferCount;      /*!< UART Tx Transfer Counter           */
  uint8_t                      *pRxBuffPtr;      /*!< Pointer to UART Rx transfer Buffer */
  uint16_t                     RxXferSize;       /*!< UART Rx Transfer size              */
  __IO uint16_t                RxXferCount;      /*!< UART Rx Transfer Counter           */
  __IO HAL_UART_RxTypeTypeDef  ReceptionType;    /*!< Type of ongoing reception          */
  DMA_HandleTypeDef            *hdmatx;          /*!< UART Tx DMA Handle parameters      */
  DMA_HandleTypeDef            *hdmarx;          /*!< UART Rx DMA Handle parameters      */
  HAL_LockTypeDef              Lock;             /*!< Locking object                     */
  __IO HAL_UART_StateTypeDef   gState;           /*!< UART state information related to global Handle management*/
  __IO HAL_UART_StateTypeDef   RxState;          /*!< UART state information related to Rx operations.*/
  __IO uint32_t                ErrorCode;        /*!< UART Error code                    */
} UART_HandleTypeDef;
```

Peripherals

NTURACING

Hardware Instance

- Hardware registers mapped to C struct and accessed by pointers to the hardware memory address

```
typedef struct {
  __IO uint32_t SR;        /*!< USART Status register,                   Address offset: 0x00 */
  __IO uint32_t DR;        /*!< USART Data register,                     Address offset: 0x04 */
  __IO uint32_t BRR;       /*!< USART Baud rate register,                Address offset: 0x08 */
  __IO uint32_t CR1;       /*!< USART Control register 1,                Address offset: 0x0C */
  __IO uint32_t CR2;       /*!< USART Control register 2,                Address offset: 0x10 */
  __IO uint32_t CR3;       /*!< USART Control register 3,                Address offset: 0x14 */
  __IO uint32_t GTPR;      /*!< USART Guard time and prescaler register, Address offset: 0x18 */
} USART_TypeDef;

#define APB1PERIPH_BASE       PERIPH_BASE
#define APB2PERIPH_BASE       (PERIPH_BASE + 0x00010000UL)
#define USART1_BASE           (APB2PERIPH_BASE + 0x1000UL)
#define USART1                ((USART_TypeDef *) USART1_BASE)
```

UART in Interrupt-Driven Mode

```
HAL_StatusTypeDef HAL_UART_Transmit_IT(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size);
HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size);

void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart);
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart);
```

- Similar API, but returns immediately, when done, `HAL_UART_TxCpltCallback()` or `HAL_UART_RxCpltCallback()` is called
- Note: The content of `pData` should not be changed until the transmission/reception is completed
  => The lifetime of `pData` should be longer than transmission
  - Global variable
  - Dellocate only after transmission is complete
  - NOT on the stack

UART Interrupts

- All interrupts are mapped to the same IRQ `USART1_IRQHandler()`

Interrupt Event	Event Flag	Enable Control Bit
Transmit Data Register Empty	TXE	TXEIE
Clear To Send (CTS) flag	CTS	CTSIE
Transmission Complete	TC	TCIE
Received Data Ready to be Read	RXNE	RXNEIE
Overrun Error Detected	ORE	RXNEIE
Idle Line Detected	IDLE	IDLEIE
Parity Error	PE	PEIE
Break Flag	LBD	LBDIE
Noise Flag, Overrun error and Framing Error in multi buffer communication	NF or ORE or FE	EIE

UART Interrupts (con'd)

- UART IRQ calls HAL handler function and it determines the cause of interrupt from the registers and calls corresponding callbacks to notify the application

```
void USART1_IRQHandler(void) {
  HAL_UART_IRQHandler(&huart1);
}
```

```
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart);
void HAL_UART_TxHalfCpltCallback(UART_HandleTypeDef *huart);
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart);
void HAL_UART_RxHalfCpltCallback(UART_HandleTypeDef *huart);
void HAL_UART_ErrorCallback(UART_HandleTypeDef *huart);
void HAL_UART_AbortCpltCallback(UART_HandleTypeDef *huart);
void HAL_UART_AbortTransmitCpltCallback(UART_HandleTypeDef *huart);
void HAL_UART_AbortReceiveCpltCallback(UART_HandleTypeDef *huart);
```

NTURACING

How Interrupt-Driven Works

- Every I/O has hardware buffer, usually only one byte long
- When empty in transmission or full in reception, an interrupt is fired to make CPU move the data to/from the buffer.
- Though `HAL_UART_TxCpltCallback()` or `HAL_UART_RxCpltCallback()` is only called when transmission is complete, internally `HAL_UART_IRQHandler()` does all the heavy lifting moving data in to and out of hardware buffer.
- The performance of interrupt-driven is not that great since CPU is interrupted every time a byte is transferred, especially when the baudrate increases.

UART in DMA Mode

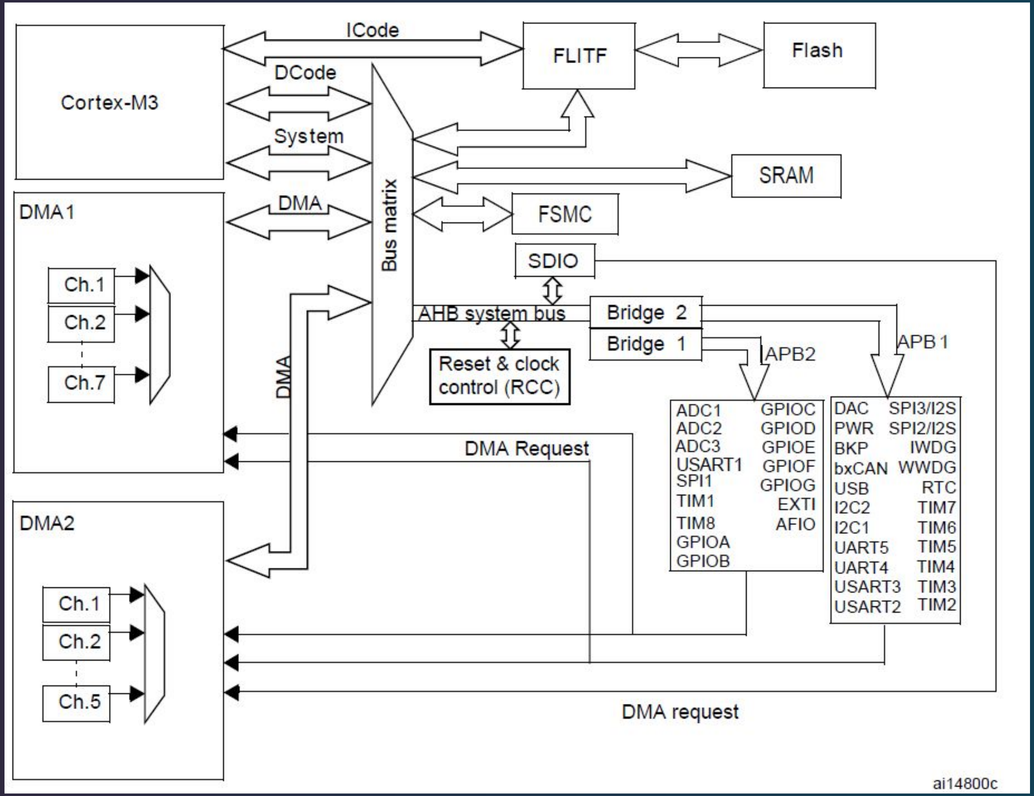
```
HAL_StatusTypeDef HAL_UART_Transmit_DMA(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size);
HAL_StatusTypeDef HAL_UART_Receive_DMA(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size);

void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart);
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart);
```

- To solve the problems of interrupt-driven I/O, direct memory access (**DMA**) can be utilized
- The same API as interrupt-driven one, and calls `HAL_UART_TxCpltCallback()` or `HAL_UART_RxCpltCallback()` when done
- Again, the content of `pData` should not be changed until the transmission/reception is completed

Peripherals

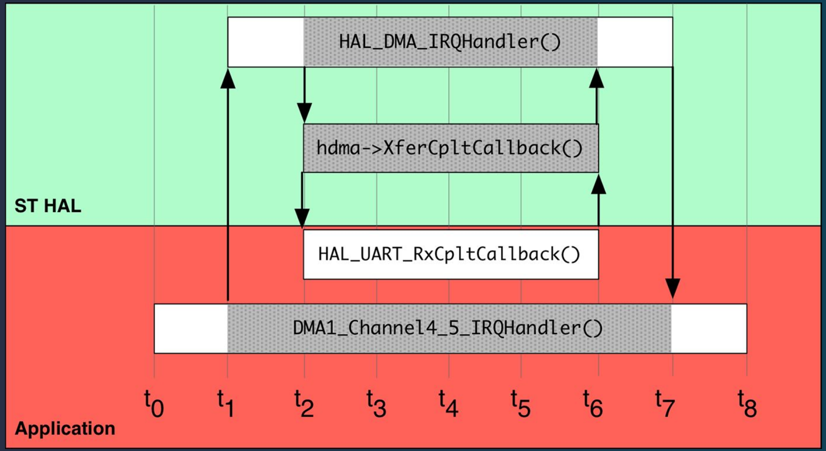
How DMA Works

- When buffer is full/empty, instead of interrupting CPU to have it move data, peripherals uses **DMA request lines** to make DMA move data instead
- DMA move data directly from/to memory to/from peripheral buffers through dedicated AHB buses
- Some DMAs could also be used to move data from memory to memory (essentially `memcpy()` without CPU)



Peripherals

How DMA Works (con'd)



- Transfer size: a word, a half-word, a byte
- Auto increment source/destanation memory address
- Priority: Low, Medium, High, Very High
- Two modes:
  - Normal: Stops when the configure amount of data is transferred
  - Circular: Start from the beginning when the configure amount of data is transferred
- Fires an interrupt after the transaction is finished to notify the application, so only one interrupt in the entire transaction
- Internally, the IRQ `DMAX_ChannelY_IRQHandler()` calls `HAL_DMA_IRQHandler()` and ultimately calls `HAL_UART_TxCpltCallback()` or `HAL_UART_RxCpltCallback()`

LAB 3: UART I

References: Ch 2, *Mastering STM32*

NTURACING

Set up VS Code for STM32CubeIDE

1. Download VS Code
2. Download gcc following this guide: https://code.visualstudio.com/docs/cpp/config-mingw
3. In VS Code, open folder on the STM32CubeIDE project directory (if using default workspace, it is C:\Users\<user_name>\STM32CubeIDE\workspace\<project_name>)
4. Install C/C++ Extension Pack extension
5. Configure the compiler to use gcc.exe
6. Add .vscode/c_cpp_properties.json
7. In C:\ST\STM32CubeIDE\STM32CubeIDE\plugins, find a directory named com.st.stm32cube.ide.mcu.externaltools.gnu-tools-for-stm32<version> and it will be <toolchain_path>

UART I

Set up VS Code for STM32CubeIDE (con'd)

8. In VS Code .vscode/c_cpp_properties.json put the following:
9. Replace <toolchain_path> with yours
10. And we are good to go

```json
{
    "configurations" : [
        {
            "name": "STM32",
            "includePath" : [
                "Core/Inc",
                "Drivers/CMSIS/Device/ST/STM32G4xx/Include" ,
                "Drivers/CMSIS/Include" ,
                "Drivers/STM32G4xx_HAL_Driver/Inc" ,
                "Drivers/STM32G4xx_HAL_Driver/Inc/Legacy" ,
                "<toolchain_path>/tools/arm-none-eabi/include" ,
                "<toolchain_path>/tools/lib/gcc/arm-none-eabi/12.3.1/include" ,
"<toolchain_path>/tools/lib/gcc/arm-none-eabi/12.3.1/include-fixed"
            ],
            "defines": [
                "STM32G474xx",
                "USE_HAL_DRIVER"
            ]
        }
    ],
    "version": 4
}
```

UART I

NTURACING

Hello World

- Normally, desktop computers does not understand UART
  => Special hardware, virtual com (**VCOM**), is used to translate UART to USB
- STLinkV3E built into NUCLEO-G474RE has such functionality
- STM32G474RE `lpuart1` is connected to it

```
char hello[] = "hello world\n\r";
HAL_UART_Transmit(&hlpuart1, hello, sizeof(hello), HAL_MAX_DELAY);
```

- To access VCOM, use putty: https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html
- The port number can be checked using device manager in Windows

NTURACING

printf() and scanf()

- Dy default, the low level functions that send to stdout, `__io_putchar()`, or receive from stdin, `__io_getchar()`, are not implemented
- Implement them as:

```
int __io_putchar(int ch) {
  HAL_UART_Transmit(&hlpuart1, (uint8_t *)&ch, 1,
HAL_MAX_DELAY);
  return ch;
}

int __io_getchar(void) {
  uint8_t ch;
  HAL_UART_Receive(&hlpuart1, &ch, 1, HAL_MAX_DELAY);
  return ch;
}
```