Programming Project (4)

Yesterday, we looked at :

- Computer Systems and MCU
- Central Processor Unit (CPU)
- Memory in a computer
- Computer Number Interpretation
- Conversion: Binary-Hex-Decimal
- ARM Cortex MCU: Block Diagram and General-Purpose I/O (GPIO)
- Relation between Register & Ports
- Configuration of GPIOs

- Exercise 5: Read & Write from I/Os
- Exercise 6: Stop watch using Timer

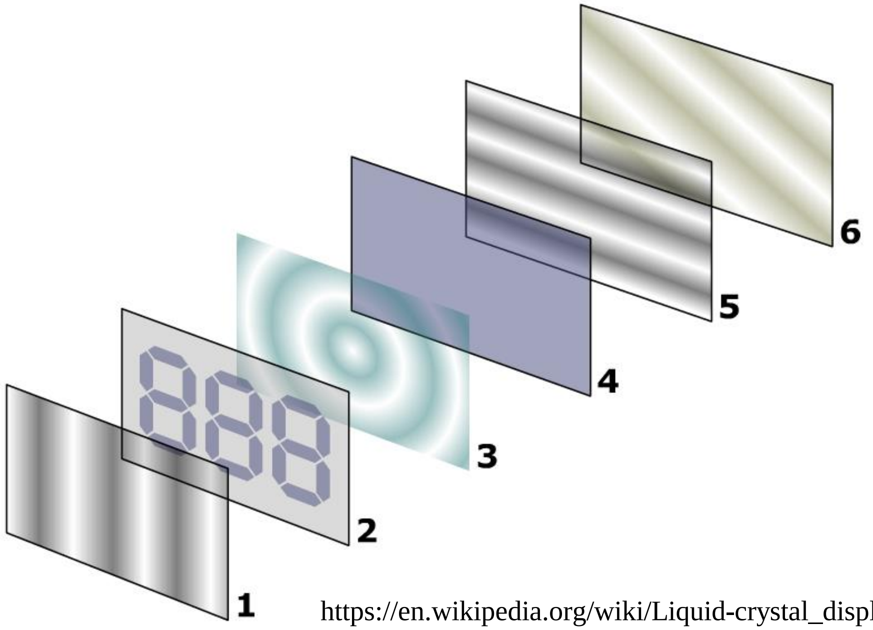
Today, we will look at:

- Repetition:
  - GPIO
  - Timer:
    - Timer Control Registers
    - Ex.: Timer 0 in ”continuous mode”
  - IRQ
- Display – LCD principle and matrix
- Serial Interfaces: SPI (& UART)
- Showing text: static & scrolling
- Digitalization of analog signals
- ADC registers on the ARM

- Exercise 7: Use LCDs for text
- Exercise 8: Using the ADC

LCD Principle of Operation

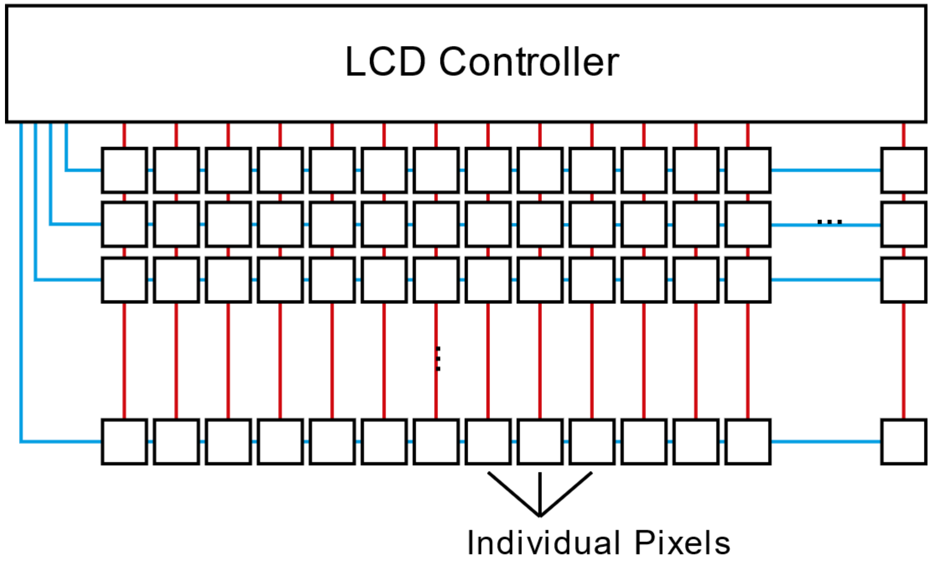
- ”Liquid crystals” polarize light.
- Electric potentials change polarization direction.
- By combining this with polarizing filters a ”light switch” can be created.



https://en.wikipedia.org/wiki/Liquid-crystal_display

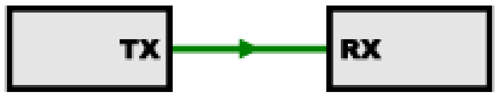
LCD Matrix Display

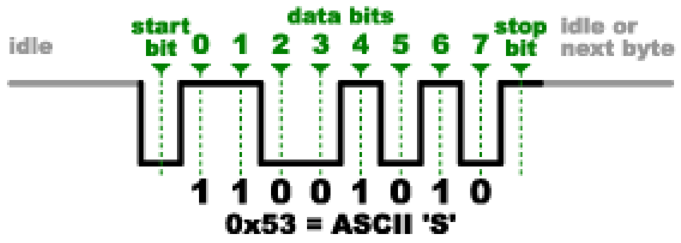
- Each pixel has its own electrode.
- The electrodes are controlled through unique combinations of horizontal and vertical control lines.
- A controller takes care of turning on/off individual pixels.
- We control the controller!



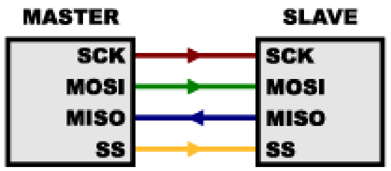
Serial Interfaces

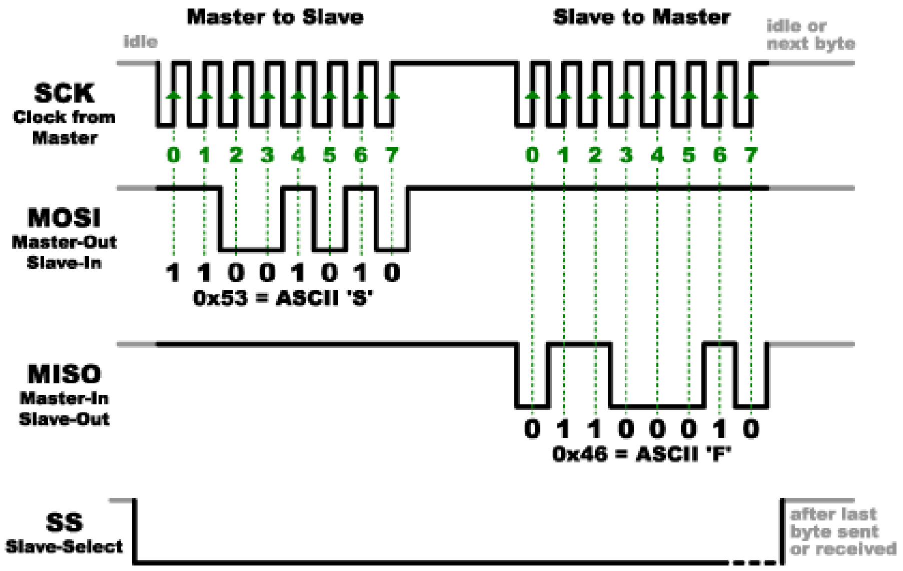
UART (Asynchronous)





SPI (Synchronous)

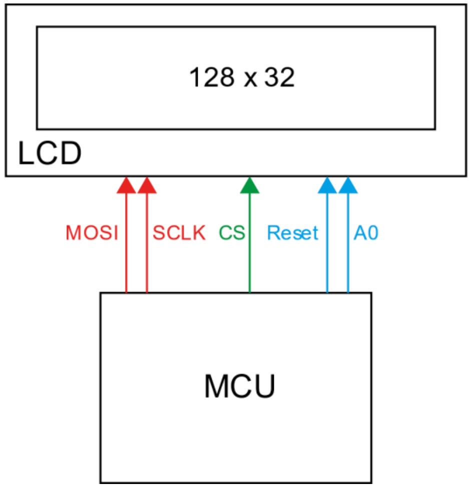




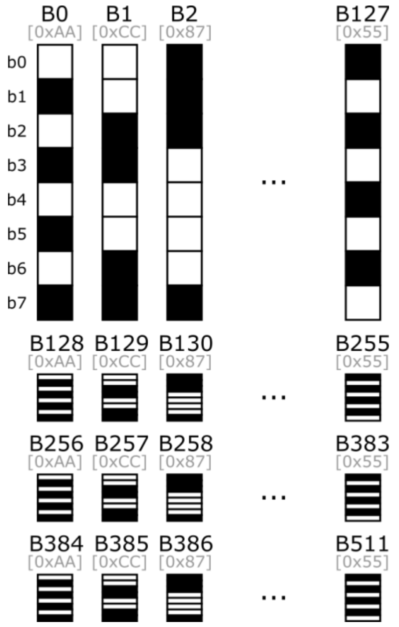
https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi

LCD SPI Communication

Hardware Interface



Software Interface



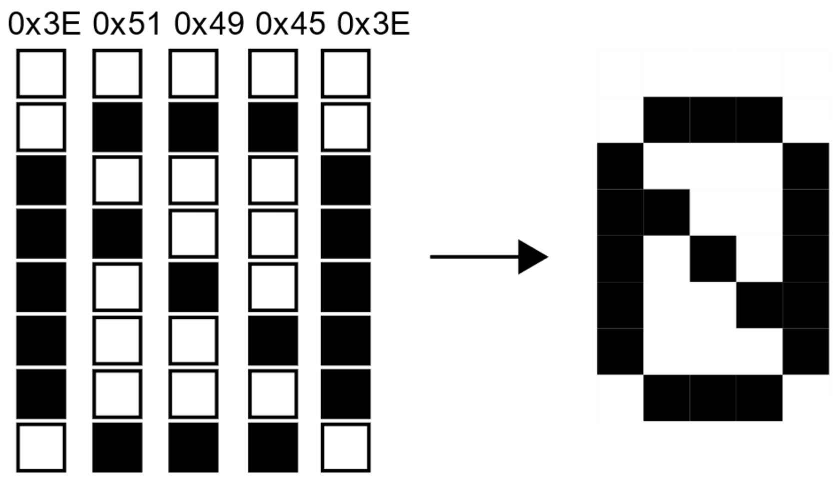
30010_io.h:

```
    init_spi_lcd(); // Initialize SPI connection
    lcd_push_buffer(&buffer); // Display buffer
```

- 512 byte buffer

Displaying Text

- Letters are stored in memory as 5 bytes per character.
- Copy character information into graphics buffer.
- Transfer buffer to LCD



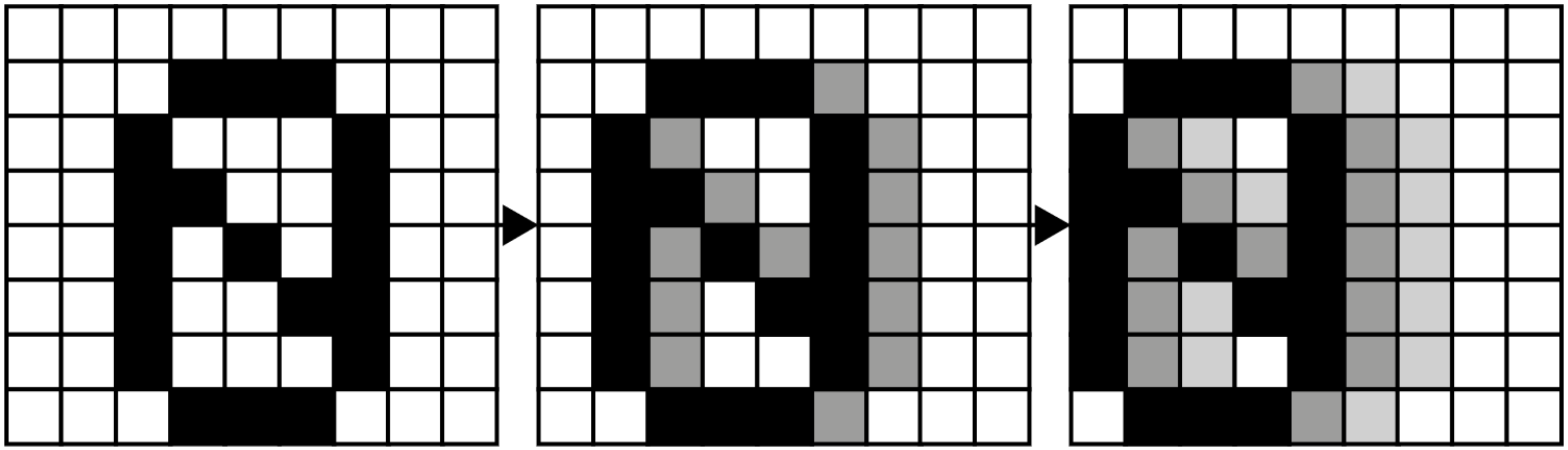
charset.h:

```
character_data[95][5] = { … }
```

Scrolling Text

- Scrolling achieved by moving everything over by one pixel.
- Doing it fast enough creates the illusion of movement.
- Use timer for speed control.
- Shifting elements of buffer / Clearing buffer and redrawing.

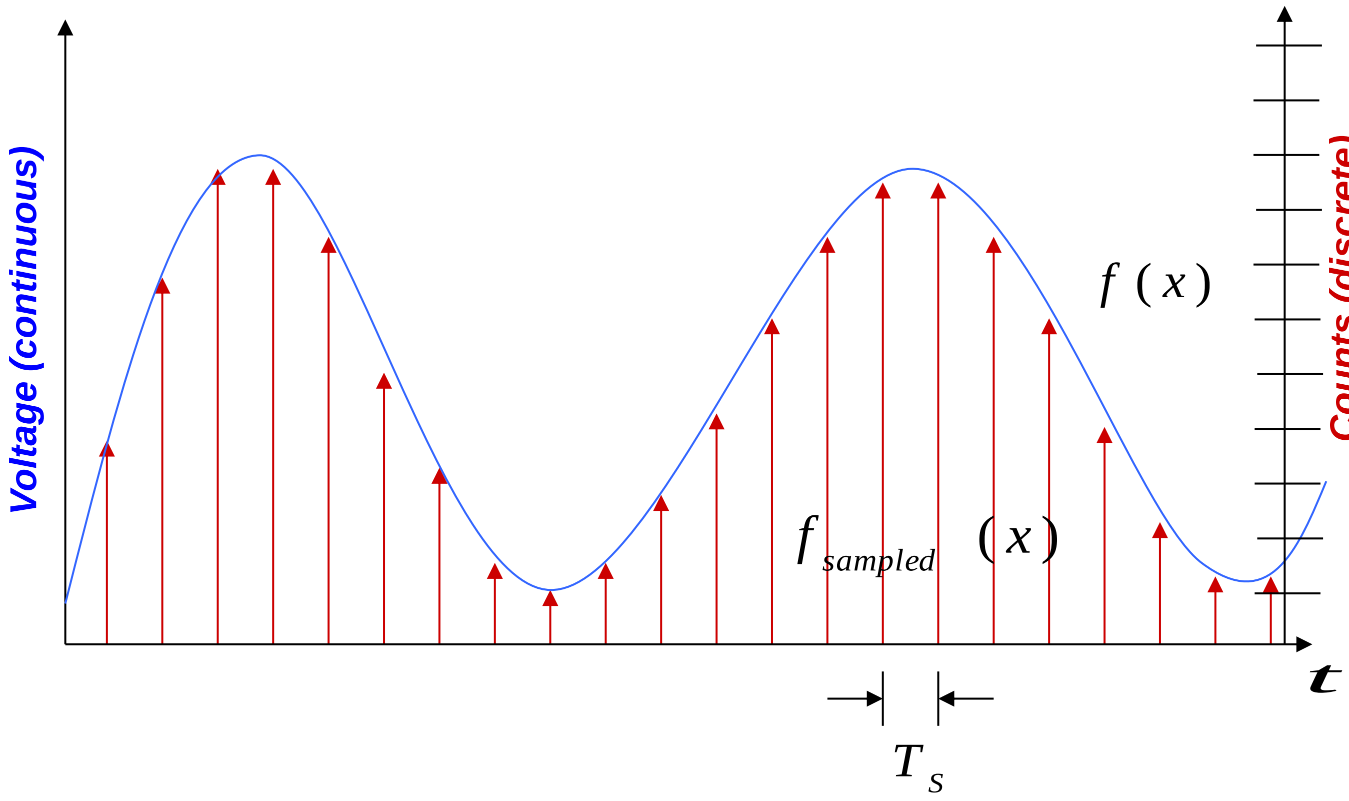
```
memset(buffer,0xAA,512); // Clear buffer
```



Digitalization of analog signals

A continuous analog signal is represented as a series of discrete values

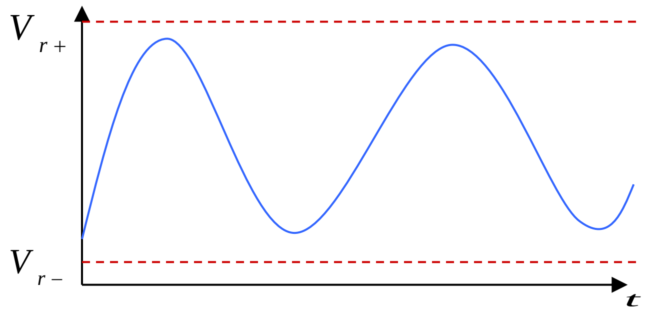
- The ADC data register produces values (counts) periodically
- The sampling frequency (fs) is the inverse of the period (Ts)



Specification of an ADC

- ADC input range: max and min Voltage that can be digitalized.

  Typical 0V-3.3V



- Resolution: number of discrete values representing the range of analog values.

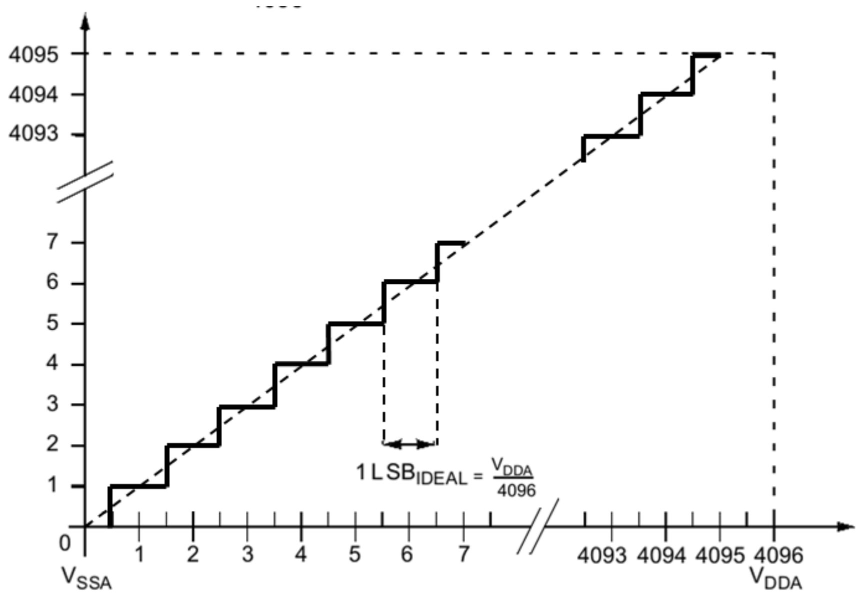
  For a 12-bit -> 2^{12}=4096 [counts]
  Range/2^{12}=step [V], 3.3/2^{12}-> 0.8mV

- Quantization Error is ½ LSB

  For a 12-bit -> 0.4mV

- Sampling frequency ($f_S > 2f_{MAX}$)

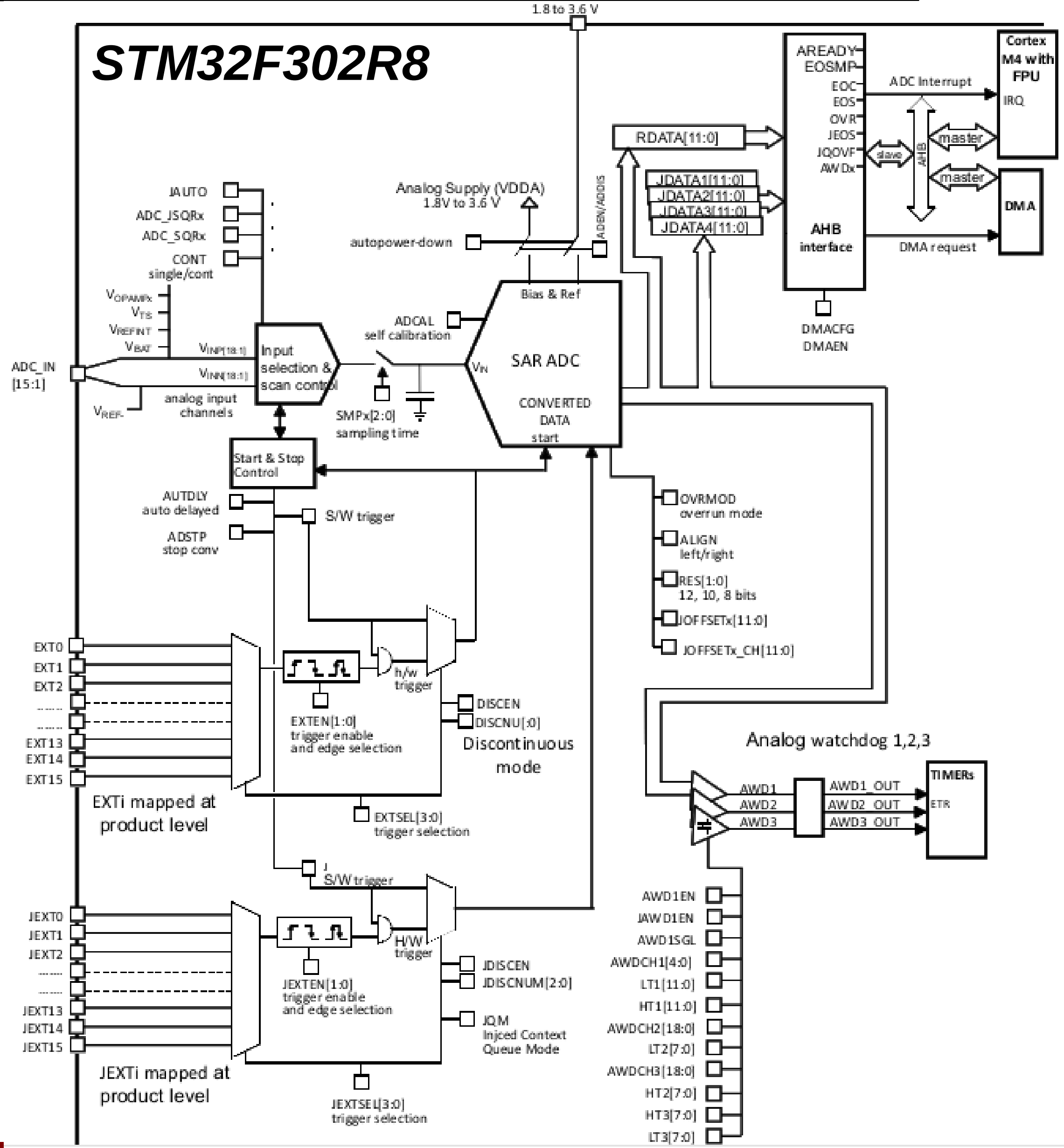
  f_S too low: loose of signal information
  f_S too high: waste of resources



ADC on the ARM µP

Characteristics of the ADC1:

- 12-bits SAR ADC
- 15 analog channels
- 0.2µs conversion time
- Single ended & diff inputs
- Self calibration
- Channels connected to:
  - Temp sensor
  - VBAT/2
  - VREFINT
  - OPAMP1
  - 11 available to user



ADC Configuration

For the ADC, the following registers need to be configured:

- 1 x 32-bit Clock configuration register 2:

  RCC_CFGR2 -> clock to ADC and PREVDIC division factor

- 1 x 32-bit AHB peripheral clock enable register:

  RCC_AHBENR -> enabling of clock

  1 x 32-bit ADC control register:

  ADC1_CR -> Cal, EN regulator, start/stor conv mode, EN AD

- 1 x 32-bit ADC configuration register:

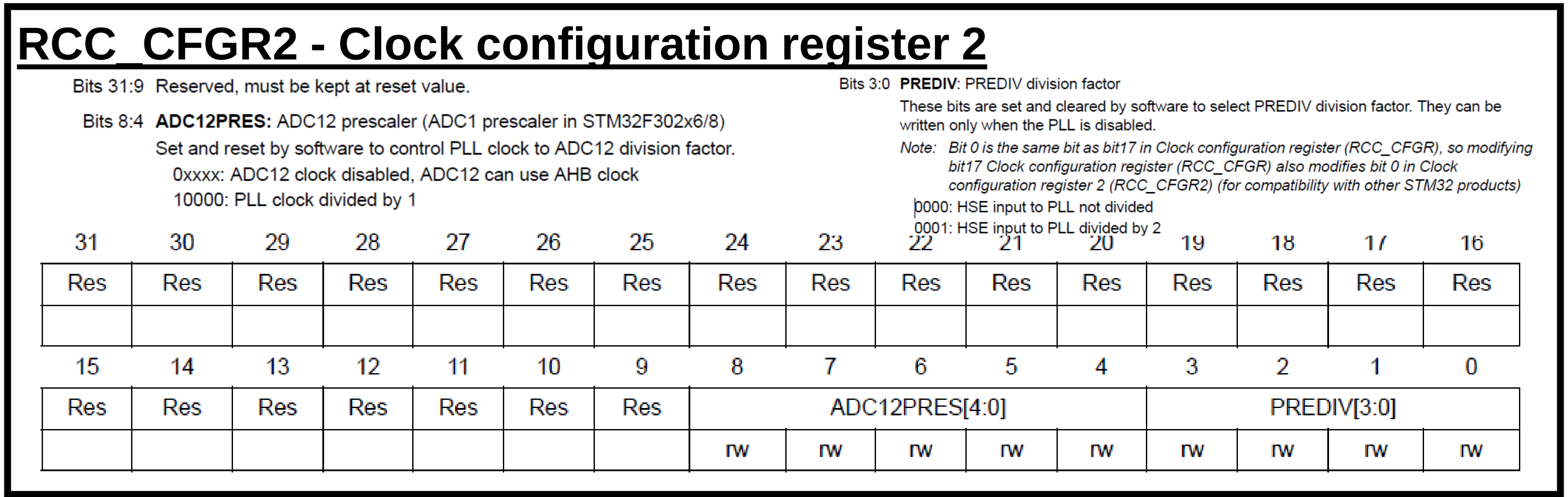
  ADCx_CFGR (x=1,2)-> EN watchdog, Cont mode, ext trigger, ADC res

- 1 x 32-bit ADC regular sequence register 1:

  ADCx_SQR1 (x=1,2)-> Configuration of sequence in conversion

STM32F302x8_Ref..._Manual, page 148-9, 138-40

Clock Registers for ADC



RCC_CFGR2 - Clock configuration register 2

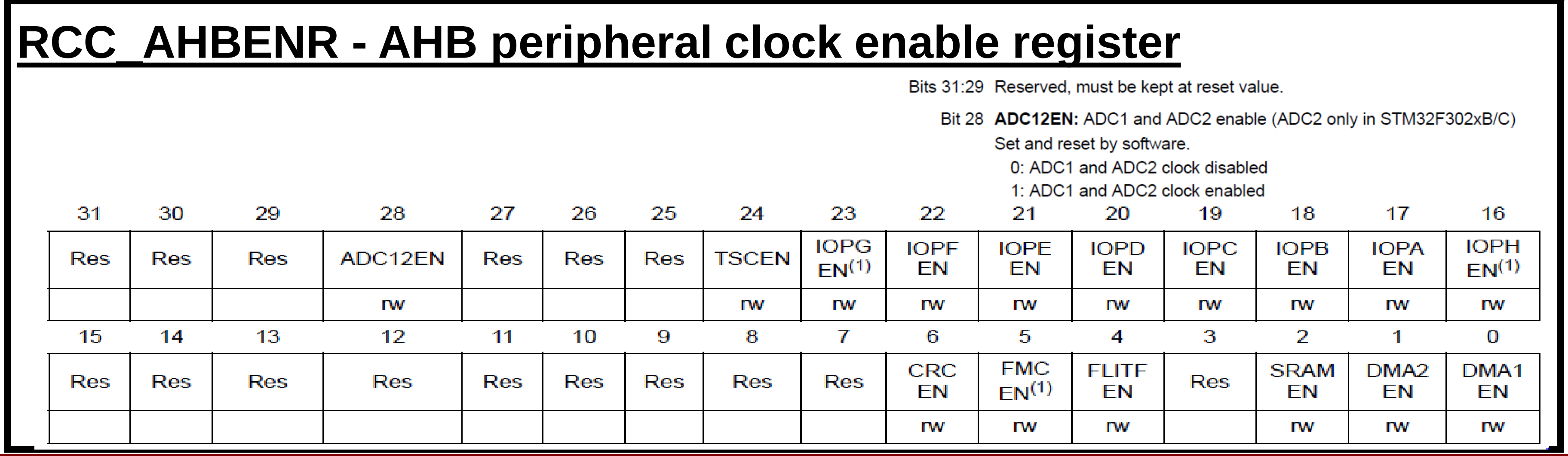
Bits 31:9 Reserved, must be kept at reset value.

Bits 8:4 **ADC12PRES:** ADC12 prescaler (ADC1 prescaler in STM32F302x6/8)
Set and reset by software to control PLL clock to ADC12 division factor.
0xxxx: ADC12 clock disabled, ADC12 can use AHB clock
10000: PLL clock divided by 1

Bits 3:0 **PREDIV**: PREDIV division factor
These bits are set and cleared by software to select PREDIV division factor. They can be written only when the PLL is disabled.
Note: Bit 0 is the same bit as bit17 in Clock configuration register (RCC_CFGR), so modifying bit17 Clock configuration register (RCC_CFGR) also modifies bit 0 in Clock configuration register 2 (RCC_CFGR2) (for compatibility with other STM32 products)
0000: HSE input to PLL not divided
0001: HSE input to PLL divided by 2

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
Res	Res	Res	Res	Res	Res	Res	ADC12PRES[4:0]					PREDIV[3:0]			
							rw	rw	rw	rw	rw	rw	rw	rw	rw



RCC_AHBENR - AHB peripheral clock enable register

Bits 31:29 Reserved, must be kept at reset value.

Bit 28 **ADC12EN:** ADC1 and ADC2 enable (ADC2 only in STM32F302xB/C)
Set and reset by software.
0: ADC1 and ADC2 clock disabled
1: ADC1 and ADC2 clock enabled

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	ADC12EN	Res	Res	Res	TSCEN	IOPG EN(1)	IOPF EN	IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	IOPH EN(1)
			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
Res	Res	Res	Res	Res	Res	Res	Res	Res	CRC EN	FMC EN(1)	FLITF EN	Res	SRAM EN	DMA2 EN	DMA1 EN
									rw	rw	rw		rw	rw	rw

DTU Space
National Space Institute

DTU

Configuration of ADC

STM32F302x8_Reference_Manual, page 360-6

ADCx_CR (x=1,2) - ADC control register

Bit 31 ADCAL: ADC calibration
Bit 30 ADCALDIF: Differential mode for calibration
Bits 29:28 ADVREGEN[1:0]: ADC voltage regulator enable
Bits 27:6 Reserved, must be kept at reset value
Bit 5 JADSTP: ADC stop of injected conversion command
Bit 4 ADSTP: ADC stop of regular conversion command
Bit 3 JADSTART: ADC start of injected conversion
Bit 2 ADSTART: ADC start of regular conversion
Bit 1 ADDIS: ADC disable command
Bit 0 ADEN: ADC enable control

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AD CAL	ADCA LDIF	ADVREGEN[1:0]		Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
rs	rw	rw	rw												

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	JAD STP	AD STP	JAD START	AD START	AD DIS	AD EN
										rs	rs	rs	rs	rs	rs

ADCx_CFGR (x=1,2) - ADC configuration register

Bit 31 Reserved, must be kept at reset value.
Bits 30:26 **AWD1CH[4:0]**: Analog watchdog 1 channel selection
Bit 25 **JAUTO:** Automatic injected group conversion
Bit 24 **JAWD1EN**: Analog watchdog 1 enable on injected chs.
Bit 23 **AWD1EN**: Analog watchdog 1 enable on regular chs.
Bit 22 **AWD1SGL**: EN watchdog 1 on a single ch. or on all chs.
Bit 21 **JQM**: JSQR queue mode
Bit 20 **JDISCEN:** Discontinuous mode on injected channels
Bits 19:17 **DISCNUM[2:0]:** Discontinuous mode channel count
Bit 16 **DISCEN**: Discontinuous mode for regular channels
Bit 15 Reserved, must be kept at reset value.
Bit 14 **AUTDLY**: Delayed conversion mode
Bit 13 **CONT**: Single / continuous conversion mode for
Bit 12 **OVRMOD**: Overrun Mode
Bits 11:10 **EXTEN[1:0]**: External trigger enable & polarity
Bits 9:6 **EXTSEL[3:0]**: External trigger selection for regula
Bit 5 **ALIGN**: Data alignment
Bits 4:3 **RES[1:0]**: Data resolution
Bit 2 Reserved, must be kept at reset value.
Bit 1 **DMACFG**: Direct memory access configuration
Bit 0 **DMAEN**: Direct memory access enable

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	AWD1CH[4:0]					JAUTO	JAWD1 EN	AWD1 EN	AWD1S GL	JQM	JDISC EN	DISCNUM[2:0]			DISC EN
	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
Res.	AUT DLY	CONT	OVR MOD	EXTEN[1:0]		EXTSEL[3:0]				ALIGN	RES[1:0]		Res.	DMA CFG	DMA EN
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw	rw

Configuration of ADC (cont.)

STM32F302x8_Reference_Manual, page 372-3

ADCx_SQR1 (x=1,2) - ADC regular sequence register 1

Bits 31:29 Reserved, must be kept at reset value.
Bits 28:24 SQ4[4:0]: 4th conversion in regular sequence
Bit 23 Reserved, must be kept at reset value.
Bits 22:18 SQ3[4:0]: 3rd conversion in regular sequence
Bit 17 Reserved, must be kept at reset value.
Bits 16:12 SQ2[4:0]: 2nd conversion in regular sequence
Bit 11 Reserved, must be kept at reset value.
Bits 10:6 SQ1[4:0]: 1st conversion in regular sequence
Bits 5:4 Reserved, must be kept at reset value.
Bits 3:0 L[3:0]: Regular channel sequence length

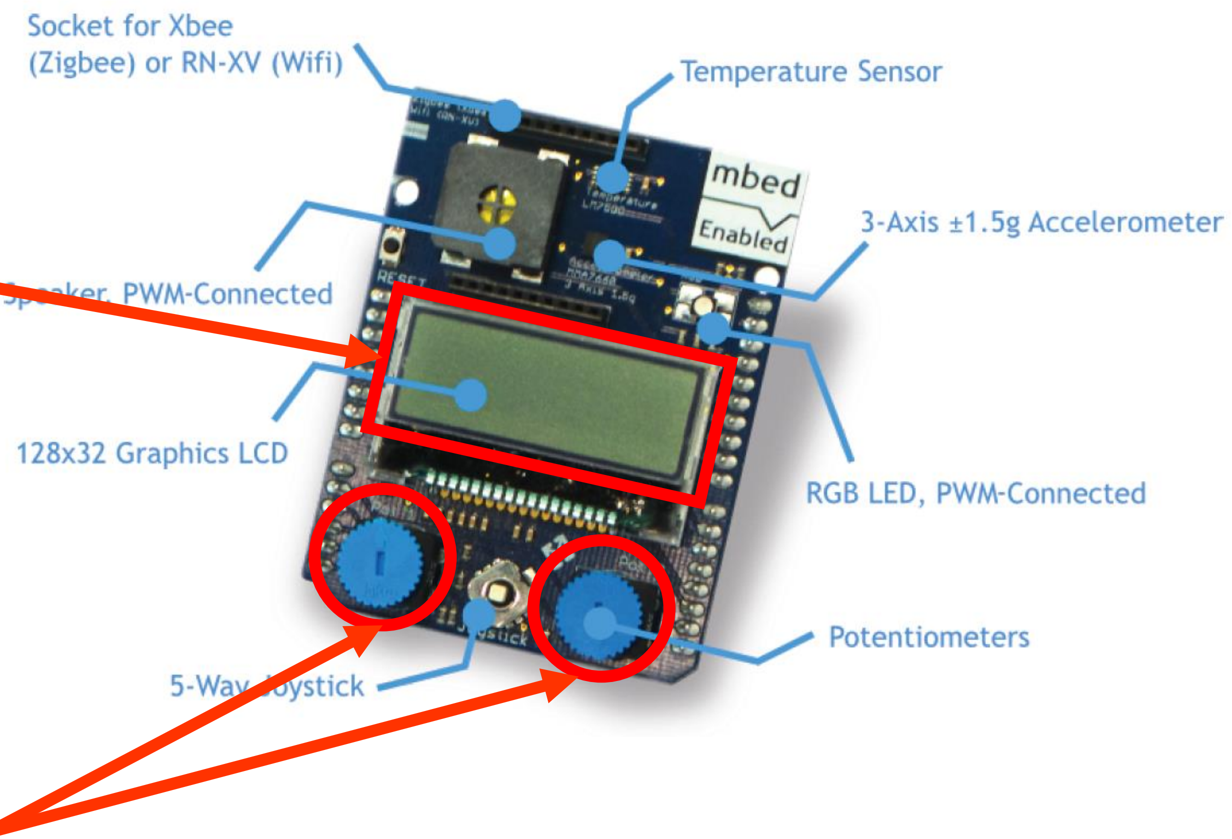
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	SQ4[4:0]					Res.	SQ3[4:0]					Res.	SQ2[4]
			rw	rw	rw	rw	rw		rw	rw	rw	rw	rw		rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ2[3:0]				Res.	SQ1[4:0]					Res.	Res.	L[3:0]			
rw	rw	rw	rw		rw	rw	rw	rw	rw			rw	rw	rw	rw

ADC functions - stm32f30x_adc.h/.c

```
void ADC_RegularChannelConfig (ADC_TypeDef* ADCx, uint8_t
   ADC_Channel, uint8_t Rank, uint8_t ADC_SampleTime);

void ADC_StartConversion(ADC_TypeDef* ADCx);

FlagStatus ADC_GetFlagStatus(ADC_TypeDef* ADCx, uint32_t
   ADC_FLAG);

uint16_t ADC_GetConversionValue(ADC_TypeDef* ADCx);
```

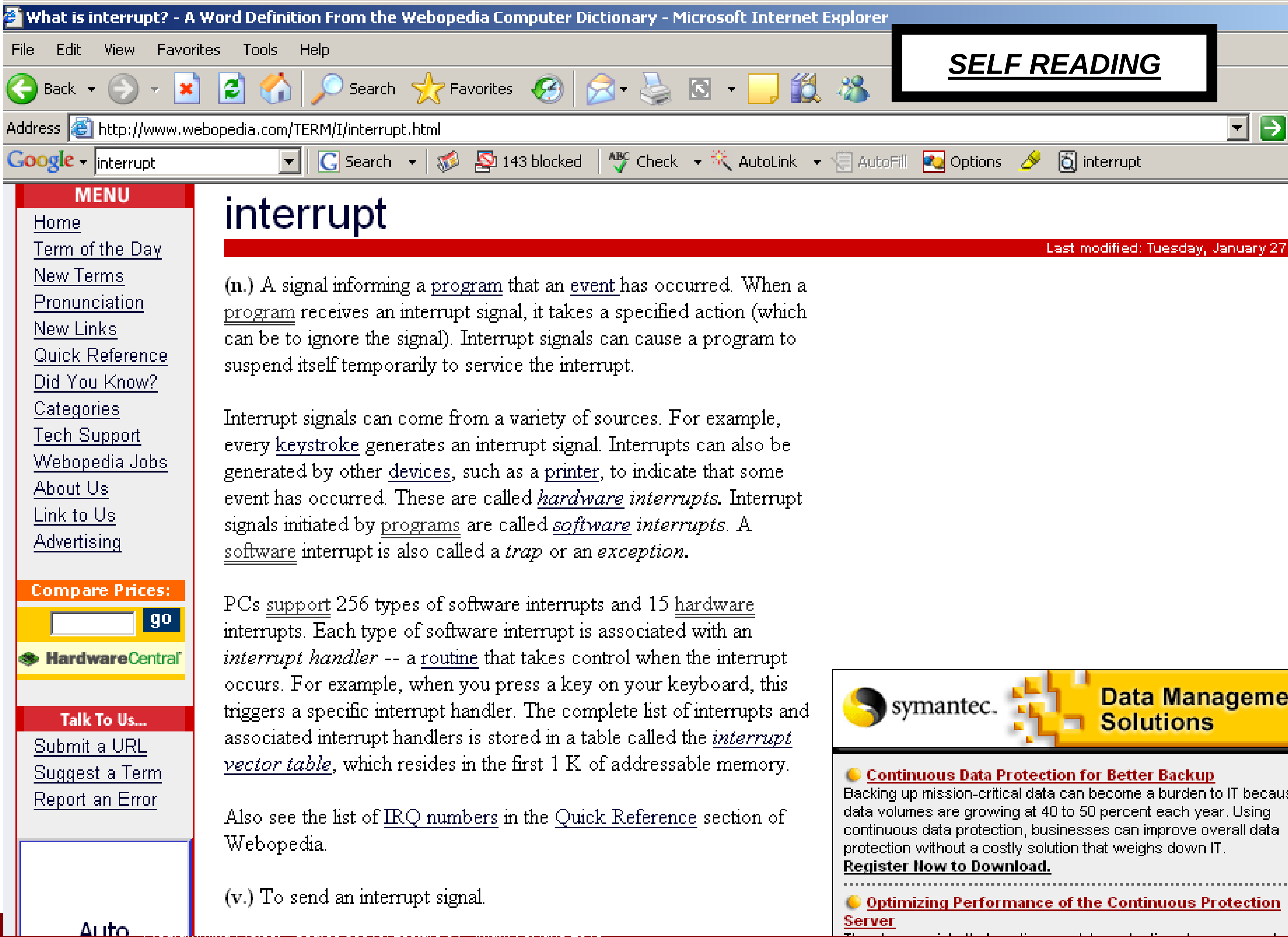
Exercise 7

- Use the LCD to print text
- Scroll the text using timers



Exercise 8

- Read analog voltage using ADC
- Print the result to PuTTY
- Print the result to the LCD display



SELF READING

What is interrupt? - A Word Definition From the Webopedia Computer Dictionary - Microsoft Internet Explorer

Address: http://www.webopedia.com/TERM/I/interrupt.html

interrupt

Last modified: Tuesday, January 27

(n.) A signal informing a program that an event has occurred. When a program receives an interrupt signal, it takes a specified action (which can be to ignore the signal). Interrupt signals can cause a program to suspend itself temporarily to service the interrupt.

Interrupt signals can come from a variety of sources. For example, every keystroke generates an interrupt signal. Interrupts can also be generated by other devices, such as a printer, to indicate that some event has occurred. These are called *hardware interrupts*. Interrupt signals initiated by programs are called *software interrupts*. A software interrupt is also called a *trap* or an *exception*.

PCs support 256 types of software interrupts and 15 hardware interrupts. Each type of software interrupt is associated with an *interrupt handler* -- a routine that takes control when the interrupt occurs. For example, when you press a key on your keyboard, this triggers a specific interrupt handler. The complete list of interrupts and associated interrupt handlers is stored in a table called the *interrupt vector table*, which resides in the first 1 K of addressable memory.

Also see the list of IRQ numbers in the Quick Reference section of Webopedia.

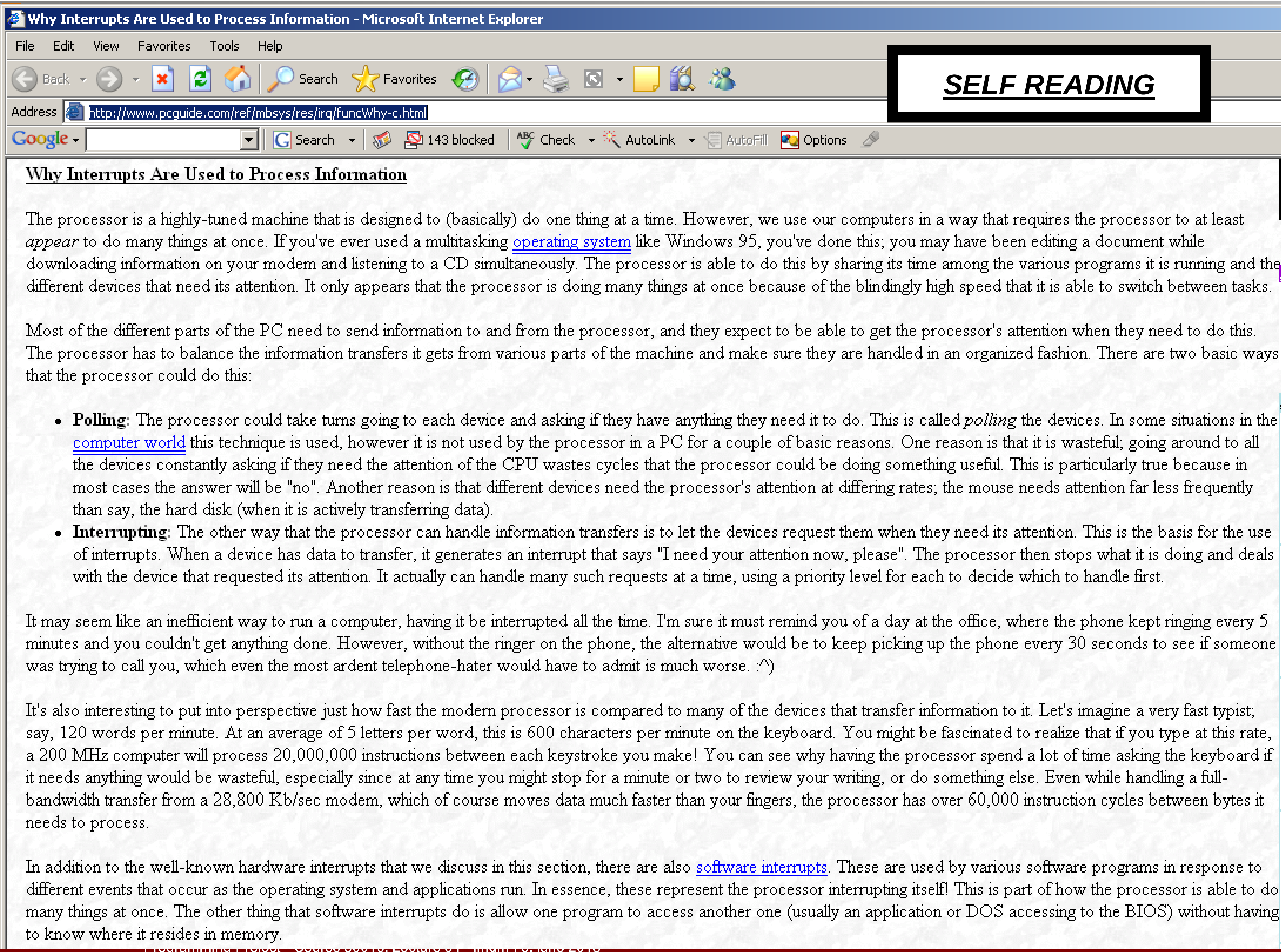
(v.) To send an interrupt signal.

Continuous Data Protection for Better Backup

Backing up mission-critical data can become a burden to IT becaus data volumes are growing at 40 to 50 percent each year. Using continuous data protection, businesses can improve overall data protection without a costly solution that weighs down IT.

Register Now to Download.

Optimizing Performance of the Continuous Protection Server


Why Interrupts Are Used to Process Information - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://www.pcguide.com/ref/mbsys/res/irq/funcWhy-c.html
Google Search 143 blocked Check AutoLink AutoFill Options

SELF READING

Why Interrupts Are Used to Process Information

The processor is a highly-tuned machine that is designed to (basically) do one thing at a time. However, we use our computers in a way that requires the processor to at least *appear* to do many things at once. If you've ever used a multitasking operating system like Windows 95, you've done this; you may have been editing a document while downloading information on your modem and listening to a CD simultaneously. The processor is able to do this by sharing its time among the various programs it is running and the different devices that need its attention. It only appears that the processor is doing many things at once because of the blindingly high speed that it is able to switch between tasks.

Most of the different parts of the PC need to send information to and from the processor, and they expect to be able to get the processor's attention when they need to do this. The processor has to balance the information transfers it gets from various parts of the machine and make sure they are handled in an organized fashion. There are two basic ways that the processor could do this:

- **Polling:** The processor could take turns going to each device and asking if they have anything they need it to do. This is called *polling* the devices. In some situations in the computer world this technique is used, however it is not used by the processor in a PC for a couple of basic reasons. One reason is that it is wasteful; going around to all the devices constantly asking if they need the attention of the CPU wastes cycles that the processor could be doing something useful. This is particularly true because in most cases the answer will be "no". Another reason is that different devices need the processor's attention at differing rates; the mouse needs attention far less frequently than say, the hard disk (when it is actively transferring data).
- **Interrupting:** The other way that the processor can handle information transfers is to let the devices request them when they need its attention. This is the basis for the use of interrupts. When a device has data to transfer, it generates an interrupt that says "I need your attention now, please". The processor then stops what it is doing and deals with the device that requested its attention. It actually can handle many such requests at a time, using a priority level for each to decide which to handle first.

It may seem like an inefficient way to run a computer, having it be interrupted all the time. I'm sure it must remind you of a day at the office, where the phone kept ringing every 5 minutes and you couldn't get anything done. However, without the ringer on the phone, the alternative would be to keep picking up the phone every 30 seconds to see if someone was trying to call you, which even the most ardent telephone-hater would have to admit is much worse. :^)

It's also interesting to put into perspective just how fast the modern processor is compared to many of the devices that transfer information to it. Let's imagine a very fast typist; say, 120 words per minute. At an average of 5 letters per word, this is 600 characters per minute on the keyboard. You might be fascinated to realize that if you type at this rate, a 200 MHz computer will process 20,000,000 instructions between each keystroke you make! You can see why having the processor spend a lot of time asking the keyboard if it needs anything would be wasteful, especially since at any time you might stop for a minute or two to review your writing, or do something else. Even while handling a full-bandwidth transfer from a 28,800 Kb/sec modem, which of course moves data much faster than your fingers, the processor has over 60,000 instruction cycles between bytes it needs to process.

In addition to the well-known hardware interrupts that we discuss in this section, there are also software interrupts. These are used by various software programs in response to different events that occur as the operating system and applications run. In essence, these represent the processor interrupting itself! This is part of how the processor is able to do many things at once. The other thing that software interrupts do is allow one program to access another one (usually an application or DOS accessing to the BIOS) without having to know where it resides in memory.