Source: http://www.fmf.uni-lj.si/~ponikvar/STM32F407%20project/

C PROGRAMMING

STM32F

Processor

1. Introduction

1.1. Motivation

The present project was started in late 2012 as a support activity in training of the use of microprocessors in physics experiments, the topic given at the University of Ljubljana, Faculty of Mathematics and Physics. There are actually two mutually supportive topics on electronic acquisition of data and presented at the faculty. One is »Acquisition and processing of signals«, the other is »The use of microprocessors in physics experiments«. The first focuses on signal acquisition with PC compatible computer using a commercial acquisition system and basic signal processing algorithms, like the ones involved in sampling, filtering and evaluation of basic signal properties. The second transfers the algorithms to a new level by introducing hardware for signal acquisition, namely microprocessors and includes a touch of FPGA based systems, and additionally introduces more advanced techniques for data processing.

In order to support the lectured theory students are introduced into the laboratory where they use either PC-based signal acquisition systems (first course) or microprocessor based units (second course). Until 2012 the microprocessor used in these systems was Texas Instruments' MSP430F1611, the 16bit unit with built-in memory, ADC, DAC, two counter/timer units ... A board using the MSP430F1611 was developed in 2007, and used since. The board's details can be found elsewhere on these pages (only in Slovenian language). However, the limitations of this processor became clear after the introduction of more sophisticated and complex algorithms in the course. The MSP430 series is optimized for low power consumption, and as such does not support either large data arrays or high speed processing. It has been demonstrated that a simple FIR filtering is limited to sampling frequencies bellow 1 kHz due to slow multiplication. Additionally, and to a regret of the designer, the lack of proper ADC input driving on the board resulted in poor resolution of the ADC, being only about 9 bits. Combining the properties of the processor used with the flops in the design limited the usefulness of the board to a demonstration level. The board was used for teaching but often with comments like “this is supposed to work as …”.

The proper solution to the problem is to introduce a new microprocessor based board of a better design. The newly designed board should at least have all the units included in the former design, but should use a contemporary processor, and should be good enough to be used in some serious work in laboratories at the faculty not just for demonstration purposes, but also for signal processing and computer interfacing. The board should be simple (EURO size, double layer at most, simple connectors for input & output signals, simple power supply, provisions to box the board into a standard enclosure, provisions to connect the board to PC computer using the USB cable, provisions to connect various sensors to the board, …).

The standard (and cheaper) solution would be to look on the market for a DEMO board which includes all units and ports required. There are several boards implementing several microcomputers available. However, the author of the first design (MSP430) is a hardware addict, and enjoys designing and building electronic circuits. There was no way to (self) convince the author into not building its own board. The board and its use will be described in the articles that will follow. Hopefully this will encourage other enthusiasts to re-build and contribute, as will the students to experiment and learn. The board may even find a way into other laboratories and experiments.

1.2. The requirements for the selected microprocessor

First of all the board should be based on a contemporary (fast) microcontroller, preferably 32 bits wide. The microcontroller should house enough RAM to hold the digitized signals (more than 10 kB as available in the MSP430F1611), and should have decent amount of ROM for program. The ROM should be "flash" type allowing it to be infinitely reprogrammable (well, not really infinite times but as experiments within a classroom are concerned infinite is a nice number to start with). There are several such microcontrollers available and are produced by many companies.

The board should be equipped with analog input lines to measure delicate analog signals with reasonable resolution. An ADC with 12 bits of resolution, supplemented with a DAC with the same resolution is nice to start with. The combination of these two allows the on-line digital processing of analog signals. The ADC can be used to sample the analog signal, and pass digital values to microprocessor for calculations. The results of calculations are outputted using the DAC back into analog world. At least two analog signals will have to be sampled, preferably more. This requires an analog multiplexer in front of the ADC, and at least two DACs since multiplexing at DAC output is not an option. The ADC/DAC requirement already reduces the number of processors to select from.

Additionally, the ADC and DAC should be fast to allow digitalization of at least stereo audio signals; this is expected to boost motivation for the use. Therefore a sub 10 μs conversion time is a necessity. Many experiments in physics are based on exact phase relationships between two or more signals. The phase relation calculations require simultaneous sampling at two or more channels and limit the usefulness of multiplexer in front of ADC. The microprocessor selected should have at least two separate ADCs to allow simultaneous sampling at two channels, and should have at least two DAC with option for simultaneous updating as well. This requirement significantly reduces the number of processors to select from.

The sampling of the ADC should be periodical, the period being defined by a quartz crystal oscillator and the divider built inside the microcontroller. Most microprocessors are equipped with such module called a timer. The timer should be able either to initiate conversion at the ADC or to trigger an interrupt to the microprocessor in many different ways, so a complex timer is preferred. If possible there should be more than one timer available. Timers are used also for counting of pulses, measuring time marks and generating digital signals. For instance a pulse width modulation (PWM) is popular for controlling the power delivered to a load. Many microcontrollers offer such timers, but less microcontrollers offer several such timers.

The ADC and DAC require periodic attention of the processor to transfer the data to and from these units. With high sampling rate the processor spends most of the transferring data from ADC and to DAC instead of calculating. A special hardware can be implemented into the microcontroller to ease the loading of the processor. A devoted memory can be used to form a circular or

intermediate buffer for data from ADC or to DAC. Alternatively, a process called Direct Memory Access (DMA) can be implemented in hardware of the microcontroller to release the processor of the data-moving task. The selected microcontroller should implement either of the two, and this also limits the number of microcontrollers available.

Sensors used in contemporary control circuit are often already equipped with ADCs and have digital outputs in form of few electrical lines called a bus. The communication over a bus conforms to a predefined set of rules described in a protocol. Protocols like I2C, SPI, RS232, USB and alike are commonly used, and the microcontroller selected should implement as many as possible of such protocols so that the programmer will not have to code the communication protocol into the software but will instead use the built-in hardware modules to implement the communication protocol without the interaction of the software. Additionally, the microcontroller should have ample number of pins devoted to the connection of sensors. This requirement also significantly reduces the number of microcontroller suitable for implementation.

Last but not least the programming of the microcontroller should be simple (low cost), since it will be used in school. The programmer should not cost more than, say, 30€. The software for writing the program should be free at least in the extent to allow the writing of school programs that are usually some kB in length. The software should have a "professional" look and feel, and should allow the compilation, linking and transfer of the program into microcontroller, as well as the debugging, single stepping, the use of breakpoints, memory and variable inspection, register peeking, manual control of microcontroller registers and output lines, and more.

1.3. The requirements for the board

The board should be EURO size and should fit into a standard enclosure. The board itself should be simple two layer design, and preferably one layer should be used as a ground plane to improve analog performance. The connections for microprocessor signals should be located at the edges of the board to allow easy access for experimenting. It is expected that the board as such will be fully exposed to students during the experimenting, and that the board anyway cannot be fully protected. After all, this is experimenting: if it is fool-proof, than it is not experimenting at all (please feel free to disagree with the above), so only a basic protection circuitry will be implemented. This opinion is based also on the past experience with the MSP430 based board, where in five years of experimenting only one microcontroller died due to a tweezers falling onto a board…

The connectors at the board should be cheap and easily repairable. Pin stripes as used for IC sockets can be used, since they are easily replaceable, and wires of correct diameter can be pushed into holes and stay there. When all wires with un-appropriate diameter are removed from a laboratory, the use of such sockets is safe. Instead of the IC pin stripes regular headers can be soldered into the board, providing a better connection for instance to the connectors on the front plate of the optional housing. The jumpers on the board should also be cheap and replaceable, so IC pin stripes or regular headers can be used.

The analog input lines can have fixed sensitivity, but must be able to allow the measurement of input signals of either polarity. The input can be made selectable by the means of a jumper: single polarity between 0 and +3 V and bipolar -3 V to +3 V. Analog output lines should be single or dual polarity as well, the selection can be done by jumpers.

The board should be equipped with some push-buttons to allow the interaction of the experimenter with the software. The board should also be equipped with a small LCD display to allow the presentation of results to the experimenter. Both pushbuttons and the display should be connected in a way that allows the connection of separate external components when the board is mounted into an enclosure.

The power supply of the board should be a universal one. A non-regulated power supply can be used, and the board should have local voltage regulator to supply the microcontroller. The local voltage regulator should be protected against reversed polarity of the power connectors. Alternatively, the power could be supplied from a USB connector plugged into a PC computer.

The board should be equipped with some type of power output. The output should be sufficient to allow the connection of at least two small DC motors or two stepper motors simultaneously. The same power output lines could be used to supply power to other loads using the PWM technique. There should be some auxiliary power amplifiers on the board, like power MOSFETs. The maximum current for each power output line should be in the range of 0.5A at least, and the transistors should be configured as open-drain circuits.

1.4. The final decision on the microcontroller and board design

Having the above requirement in mind a board was designed and built. It is based on ST microcontroller STM32F407 in 100pin LQPF package. The package was selected for simple soldering. The STM32F407 is a 32 bit ARM CORTEX-M4 architecture microcontroller with 512 kB flash memory and 192 kB of RAM. It runs with internal clock of up-to 168 MHz, and has lots of goodies built in, among them all the required ones. The details of the microcontroller are available at: http://www.st.com/internet/mcu/product/252149.jsp . Yes, the author is aware that there are numerous microcontrollers available that fulfill the requirements. However, the author opted for this particular chip. The decision was not based on former experience with this processor or this family of processors. The decision was not based on options not stated in the requirements. The decision was mostly based on a review of specifications of the available components and the support provided by companies, as well as the price for the finalized board. If the author would spend more time deciding then the decision might have been a different one, and the author is not willing to spend time arguing he made an optimal decision.

The basis for the project was the STM32F4DISCOVERY board, sold by ST for a small price of about 14€ including the programmer (yes, this board also eased the decision on the chip). The initial idea was to use this board and make it sit in a “piggyback” (whatever this means literally) position on a homemade board with the required connectors and power drivers. The DISCOVERY board, regrettably, proved to offer poor analog performance due to its poor layout. The DISCOVERY board can still be used as a programmer since it is cheap and well supported. It can also be used where analog performance is not critical. However, it is without the protective casing, and one might prefer to use regular programmer like ST-link/v2, priced at about 25€ (http://www.st.com/internet/evalboard/product/251168.jsp).

The STM32F407 uses 3.3 V power supply at a current depending on the peripherals used and the clock speed. When only the CPU is running at full speed of 168 MHz, the current consumption is about 50 mA; with all peripherals enabled the current consumption increases to about 100 mA. Details are given at the chip’s datasheet.

The STM32F4xx series has 5 V tolerant inputs. This means that regardless of the power supply a signal between 0 V and 5 V can be connected to any input of a microcontroller and no harm will be done. If one exaggerates and connects a pin to a signal outside of these limits, then the consequences will have to be corrected. The chip seems to be considered protected against reasonable amounts of static electricity, since also the STM32F4DISCOVERY board does not implement any protection against the static electricity, and there are no noticeable warnings to the potential user. However, regular protection against static electricity is welcomed, and should be followed. The details on “electrical hardness” can be found in the STM datasheet.

The allowed regular output current on all digital outputs from STM32F4xx series is limited to few mA as in any digital circuit. In order to protect the chip against the over-current when input voltage exceeds the normal 0 V to 5 V range a series resistor is inserted in all digital lines that are available to the user. The prototype uses resistors of either 100 Ω or 33 Ω, therefore an over-voltage of 0.5V (for 100Ω resistor in series) will do no harm (guaranteed, the datasheet allows 5 mA of “injected” current). Where this is considered not sufficient, a higher value resistor could be inserted bearing in mind that such resistor not only limits the regular output current but also the speed of transition from one logical level to the other and may influence the behavior of the circuit.

The analog input circuitry will be based on the use of operational amplifier in buffer configuration. Fast, rail-to-rail IO chips will be used. The power supply for these chips will be made selectable by the means of jumpers. One may use either 0 V / +3.3 V power to guarantee the safety of the microcontroller input pins. However, such power supply of the operational amplifiers reduces the range of input voltages to the ADC within the microcontroller. Some tens of mV are stripped at either supply line, limiting the ADC range from about 50 mV to 3.25 V. Where this cannot be tolerated, the power supply for the buffer chips can be increased to ±5 V assuring the full 0 V to 3.3 V at the input to the ADC. However, with this the microcontroller can be harmed by the overvoltage at microcontroller pins, so a double diode should be used to divert the over-current to either power supply. The additional ±5 V power supply can be provided by a local DC-DC switching regulator.

The analog input lines should be selectable to allow either the measurement of positive signals or bipolar signals. A simple voltage divider will be placed in front of the buffer amplifier to allow the selection. The analog output lines will be buffered using a operational amplifier. The same as stated for the analog input buffer regarding the power supply and limits also applies here.

The digital output lines will be as many as applicable, and will be available as chunks of microcontroller ports. The chunks will be ordered sequentially and will be as big as possible. The connectors where these chunks are available will be expanded by several ground and power pins to allow the connection and power supply of additional circuit. All digital output lines will be protected by a series resistor of the selected value, see above. The same lines can also be used as inputs.

The LCD used will be a standard 16 characters by 2 lines display. It will be mounted onto the board using long spacers to allow components to be placed underneath. The LCD will be connected to the board using a header; the data transfer to the LCD will use a standard bus but will only be 4 bits wide. The placement of the LCD should not reduce the accessibility of any vital component.

There will be six pushbuttons on the board. They are provided with pull-up / pull-down resistors to ease the use for those not aware of pull-ups / pull-downs built into the microcontroller. Additionally, there will be a provision to connect external keyboard using a standard header

There will be some LEDs on the board. These will be connected to spare outputs of the microcontroller to allow the signaling of vital microcontroller stated as defined in the user software.

There will be two driver chips for power output signals. The selected units are L293D, which host two full H-bridge configurations. They can provide up-to 1A output current, and can be supplied with maximum of 25V. The chips have protective diodes built-in, and can be paralleled in the case of emergency. The control signals for the drivers are taken from the PWM outputs of timers built inside the microcontrollers, but can be controlled by the software as well. The same control signals are also available at a connector to be used by an alternative power driver if needed.

There will be a mix of connectors to support various communication protocols: I2C, SPI, RS232 (TTL and ±12V versions). There will be a provision for USB connection to the personal computer. The last can be optically isolated to reduce the interference from the computer.

There will be 4 MOSFET power transistors on the board, each with separate input and output. The transistor are BSP296 type, allowing the switching of up-to 1A at up-to 500V (not both limits simultaneously, please; see transistor specifications).

This sums up the requirements and the decisions made, and now it is time to see the board.

2. The STM32F4-Discovery BaseBoard

The description of the extension board based on the STM32F Discovery is given in this chapter. Full schematic diagrams and PCB layouts are presented, including figures on signal connections.

2.1. The processor board

There are many different kinds of microcontrollers available, and all are optimal from a certain point of view.

The experiments that are to be performed in this course require a microcontroller that is capable of typical data processing tasks in real time; the processor must be fast, preferably 32 bit type. Analog signals will be used as a source of information, therefore the microcontroller must be capable of converting analog signals to digital values and back. Additionally, the microcontroller should have provisions for typical buses as used in contemporary interfacing, like I2C, SPI, and RS232. An exchange of data between the microcontroller and a personal computer must be feasible.

The microcontroller is to be programmed using a "free" version of compiler, the preferred programming language is "C". No operating system will be used. The hardware needed for programming of the microcontroller should be freely available.

Due to its convenient properties and low price the STM32F4-Discovery demo board is selected for this course. The demo board can be obtained for as low as 15€, and includes the microcontroller STM32F407VGT (32bit, ARM Cortex-M4F core, 1MB Flash memory, 192kB RAM), some interfacing hardware, and the programmer. The board is powered through an USB connector from a host personal computer. Most of the microcontroller pins are available at two header connectors on the board. The board can be used without any additional hardware, and the detailed description is given at:

http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/PF252419 (as of Nov. 2013)

2.2. The extension - BaseBoard

The STM32F4-Discovery board hosts a microcontroller with most of its pins exposed at the header connectors, Fig. 2.2. In order to prevent damage caused by either static electricity or experimenting faults a BaseBoard has been prepared. This board is a EURO size printed circuit board with header sockets that accept headers of the STM32F4-Discovery board, and includes protection circuits and additional hardware to improve the interfacing experience, see Fig. 2.2.13 for signal names at header pins.

2.2.1. The BaseBoard power supply, Fig. 2.2.1

The power to run STM32F4-Discovery board is normally supplied from a USB cable connected to a personal computer (PC). However, there are times when the board is to be used without the connection to the PC, and in such cases the board needs an alternative power supply. This can be any source capable of providing a voltage between 8V and 25V, 0.25A min. This source can be connected to the BaseBoard using pads B200 and B201; there is a protective diode D270 to prevent damage due

to the incorrect polarity, and a LED D242 to signal the presence of the external power source. The BaseBoard includes a regulator LM7805 (U800) to reduce the applied voltage to 5V as required to power the Discovery board. There are several decoupling capacitors added on the board. The input voltage is used also for the driving of the power outputs from the BaseBoard.

The regulated +5V voltage should not be applied to the Discovery board simultaneously with a voltage from a USB cable. A jumper J802 is used to disconnect the external power supply when the USB cable is attached to the Discovery board.

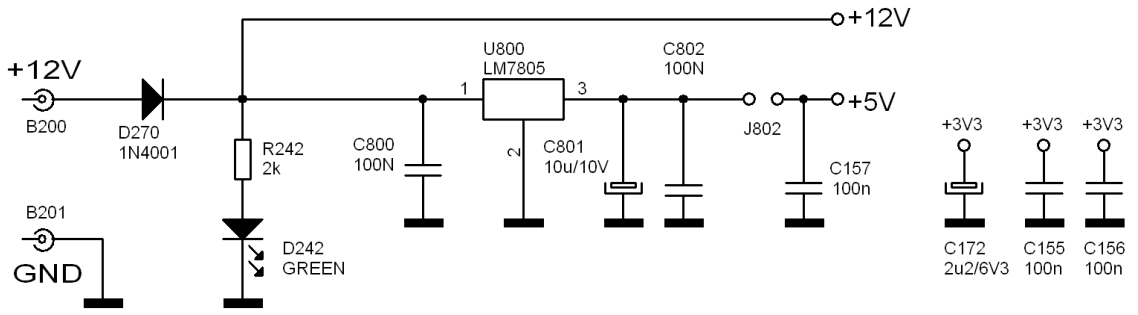


Fig. 2.2.1: The BaseBoard power supply

2.2.2. The switches, Fig. 2.2.2

There are four push-button switches S370 to S373 mounted on the BaseBoard, these can be used by the user program as required. Switches are pull-up type, and are connected to the microcontroller port E, pins PE3 to PE6. The user software should declare these pins as inputs with active pull-down. Resistors R370 to R373 are inserted between switches and the microcontroller to prevent damage to the microcontroller in case of false programming.

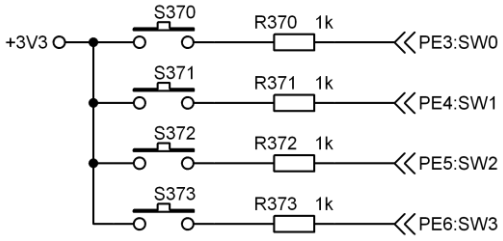


Figure 2.2.2: The switches

2.2.3. The alphanumerical liquid crystal display (LCD), Fig. 2.2.3

An alphanumerical LCD is mounted on the BaseBoard. The LCD is a standard 16 character – 2 line display, and can be accessed through four data (PD3:LCD_D3/7 to PD0:LCD_D0/4) and two control lines (PD6:LCD_E, PD7:LCD_RS), all connected to port D. The contrast of the displayed characters can be adjusted using the trimmer P350.

The LCD is powered by +5V, but accepts 3V signals from the microcontroller. The writing on the display requires rather complex routines, and these have been prepared in advance as a file to be included in the user program ("LCD2x16.c").

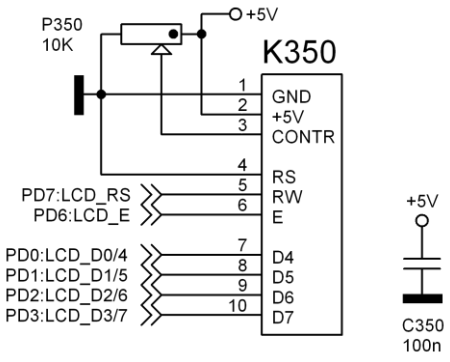


Figure 2.2.3: The alphanumerical LCD

2.2.4. The interface I2C, Fig. 2.2.4

The interface bus I2C uses two signals SCL and SDA. The hardware to manage these two signals is built into the microcontroller (block number 1 for I2C is used), and the two signals are available at port B, pins PB6:I2C1_SCL and PB7:I2C1_SDA. The bus signals are available at connector K460 at the edge of the BaseBoard.

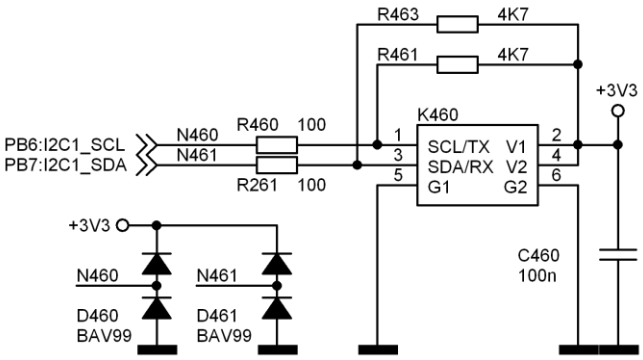


Figure 2.2.4: The interface I2C

Two resistors R461 and R463 are added at the BaseBoard as a pull-up resistors as required by the I2C standard. Two resistors R460 and R261 in connection with diodes D460 and D461 are inserted as protective elements.

2.2.5. The interface SPI, Fig. 2.2.5

The interface bus SPI uses four signals SCK, MISO, MOSI and SEL. Two SPI connectors are implemented, therefore two select signals SEL are needed. The hardware to manage all these signals is built into the microcontroller (block number 2 for SPI is used), and signals are available at port B,

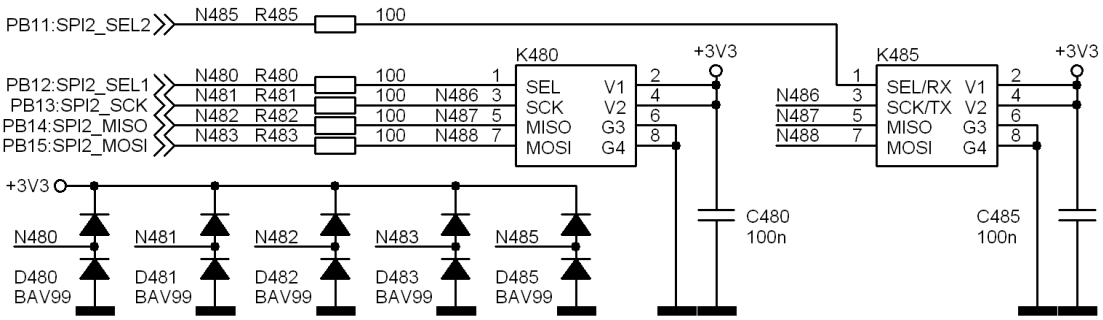


Figure 2.2.5: The interface SPI, two times

pins PB13:SPI2_SCK, PB14:SPI2_MISO, PB15:SPI2_MOSI. The two select signals are available at PB12:SPI2_SEL1 and PB11:SPI2_SEL2. The bus signals are available at connectors K480 and K485 respectively.

Five resistors R480 to R485 together with five diodes D480 to D485 are added as protective elements to reduce the chance of harming the microcontroller.

2.2.6. The interface RS232, Fig. 2.2.6

The interface bus RS232 uses two signals RX and TX. The hardware to manage these signals is built into the microcontroller (block number 3 for serial communication is used here), and signals are available at port D, pins PD9:USAT3_RX and PD8:USART3_TX. The signals are available at connector K600 at the edge of the BaseBoard. The RS232 signals here are TTL compatible (0V to 3V), and should NOT be connected to the RS232 of a normal PC. This version of RS232 signals are intended to be connected to the "USB to serial cable FTDI TTL-232R-3V3" (or similar).

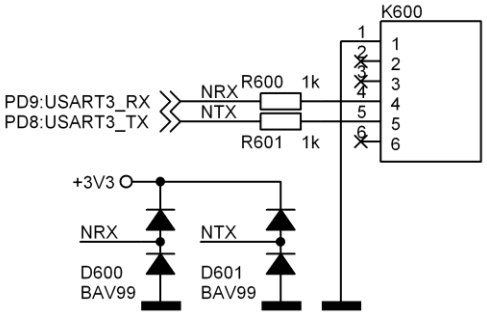


Figure 2.2.6: The interface RS232

Two resistors R600 and R601 in combination with two diodes D600 and D601 are used as protective elements.

2.2.7. General purpose digital input/output, port E, Fig. 2.2.7

Eight lines of port E (PE8 to PE15) are available for digital interfacing, these are available at the connector K440. The connector additionally offers a power supply of +3V and a power supply of +5V.

Resistor R440 to R447 in combination with diodes D440 to D447 serve as protective elements.

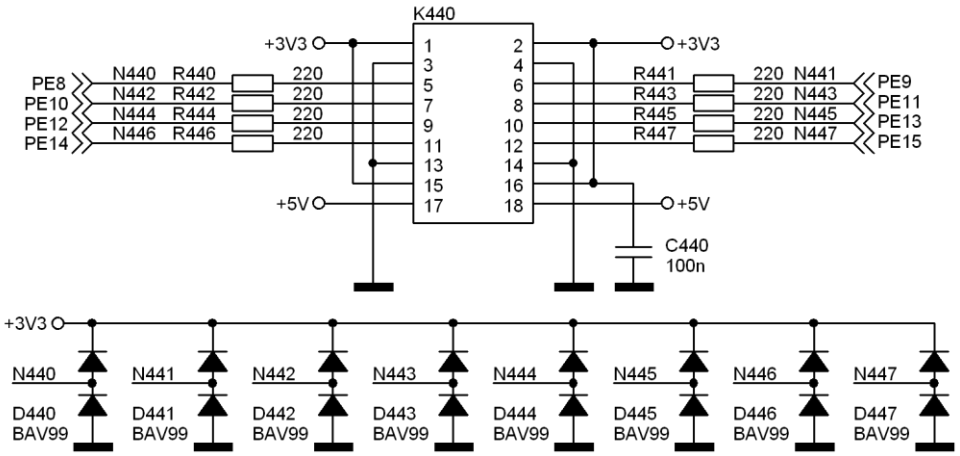


Figure 2.2.7: General purpose digital input/output, port E

2.2.8. Counters or general purpose digital input/outputs, Fig. 2.2.8

The microcontroller houses several counters/timers. The connections to some of them are available at the connector K406. The same can be used as general purpose digital input/output when counters are not required.

Resistors R406 to R411 in combination with diodes D406 to D411 serve as protective elements.

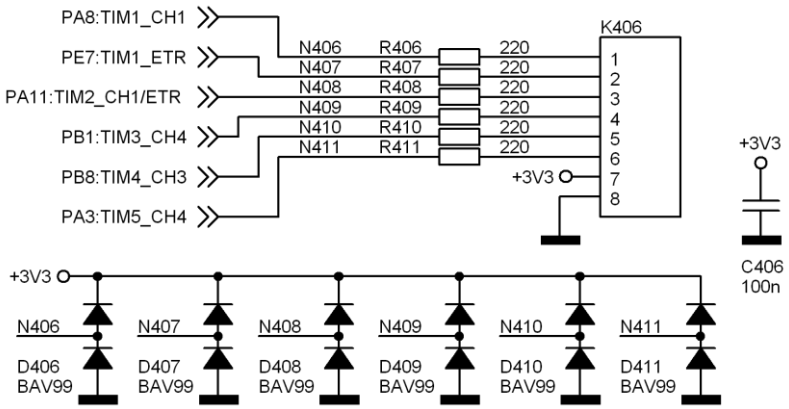


Figure 2.2.8: Counters or general purpose digital input/outputs

2.2.9. Analog output, Fig. 2.2.9

The microcontroller hosts two DACs (12 bit resolution), their outputs are available at port A, PA4:DAC1_OUT and PA5:DAC2_OUT. These pins of port A must be declared properly to serve as analog outputs. The signals from these pins are buffered and filtered using two operational amplifiers and passive RC filters, and are finally available at connector K300. The range of output voltages is from 0V to +2.5V, the output current is limited by the internal circuitry of the operational amplifier U320.

The pins PA4 and PA5 are used on the STM23F4-Discovery board to drive digital inputs of peripheral chips. In order to fully utilize the properties of DAC, the PCB traces to these peripheral chips should be cut.

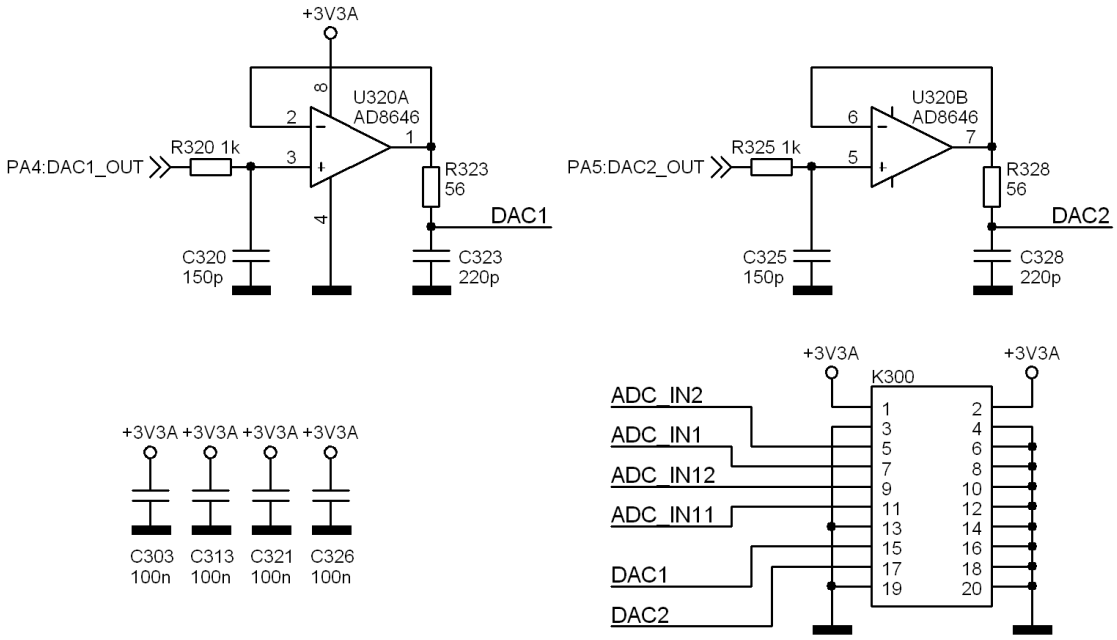


Figure 2.2.9: Analog output

2.2.10. Analog input, Fig. 2.2.10

The microcontroller hosts three ADCs (12 bit resolution each), and three multiplexers to connect signals from several pins of the microcontroller to each of them. The BaseBoard utilizes four pins to measure external analog signals, these are PA1:ADC123_IN1, PA2:ADC123_IN2, PC1:ADC123_IN11, and PC2:ADC123_IN12. These pins must be initialized to analog mode prior to the use of the ADC function, and external analog signals can be connected to connector K300.

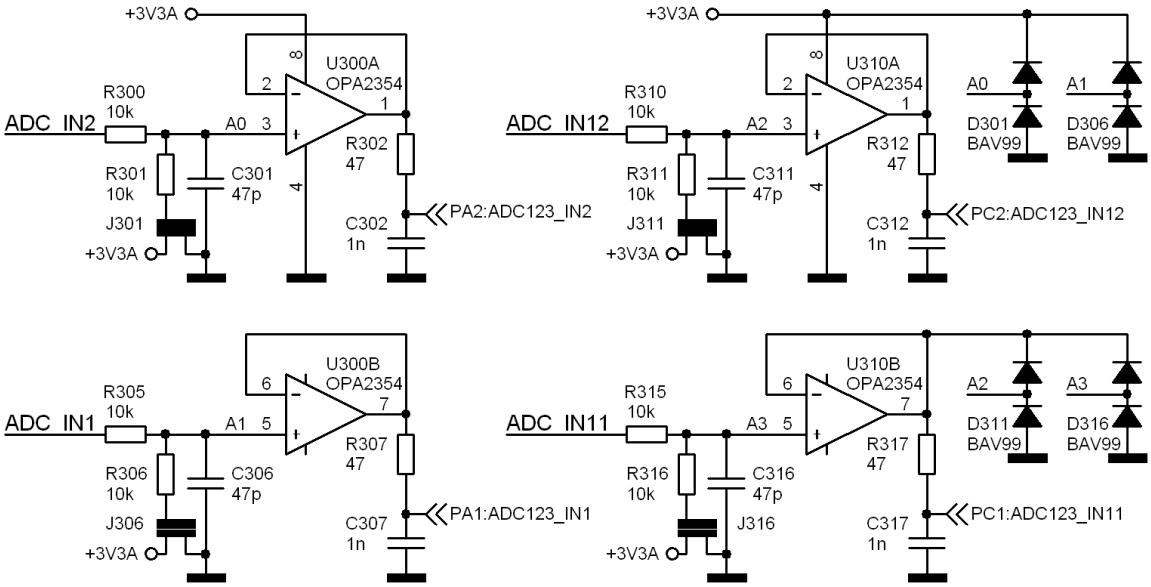


Figure 2.2.10: Analog input, the connector is given in Fig. 2.2.9

Input signals are fed to four identical stages before being passed to the microcontroller. Each stage consists of a voltage divider (resistors R300, R301, capacitor C301 and jumper J301 for the first stage). The divider is followed by a unity gain connected operational amplifier U300A and an RC low-pass filter (R302 and C302). A double diode D301 is connected to the non-inverting input of the operational amplifier, and serves as a protective element.

The input range depends on the position of the jumper (J301 for the first stage):

- Jumper not inserted: range is 0V to +3V3 (Vref of the microcontroller)
- Jumper shorts R301 to ground: range is 0V to two times +3V3 (2 x Vref of the microcontroller)
- Jumper shorts R301 and +3V3: range is -3V3 to +3V3.

Both RC time constants limit the input frequency range to 1MHz.

2.2.11. Power digital outputs, Fig. 2.2.11

Two lines of port C (PC14 and PC15) are used to drive MOSFET transistors Q450 and Q451. The transistors are connected as pull-down switches where the drains are available at the connector K451, Fig. 2.2.11. The same connector hosts ground and power supply +5V and +12V. The maximum current for the MOSFET is 1A.

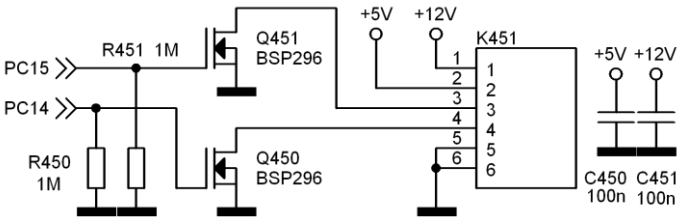


Figure 2.2.11: Power digital outputs

2.2.12. Motor driver, Fig. 2.2.12

Five lines of port C (PC6 to PC9 and PC11) are buffered using an integrated circuit U500, and can be used to drive a stepper motor or two DC motors. The circuit U500 must be enabled prior to the use by setting its enable input (PC11) high. Buffered signals PC6 to PC19 then appear at connector K500, Fig. 2.2.12. The circuit U500 is supplied by the external power supply +12V applied at B200. The maximum output current is 800mA.

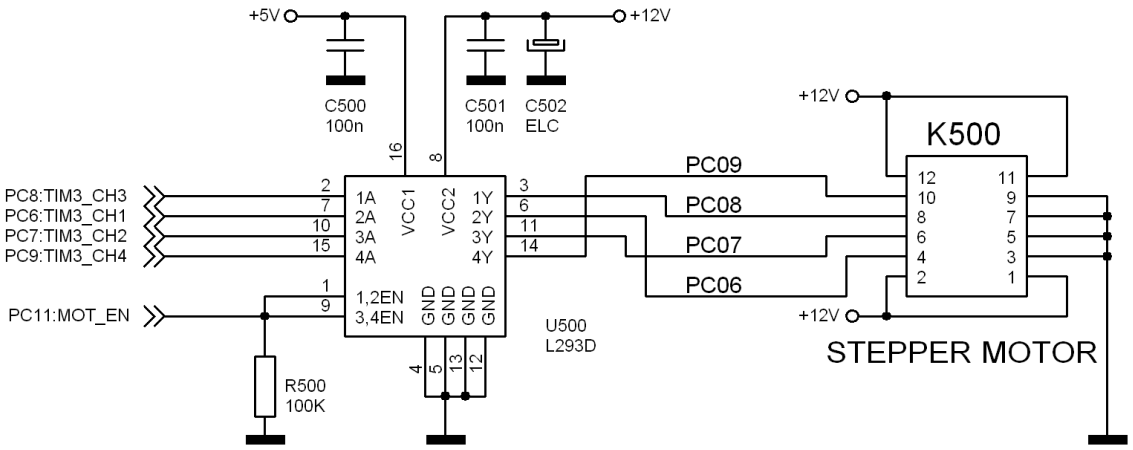


Figure 2.2.12: Motor driver

2.2.13. STM23F4-Discovery header signals

The signals at the headers of the STM23F4-Discovery board are given in Fig. 2.2.13.

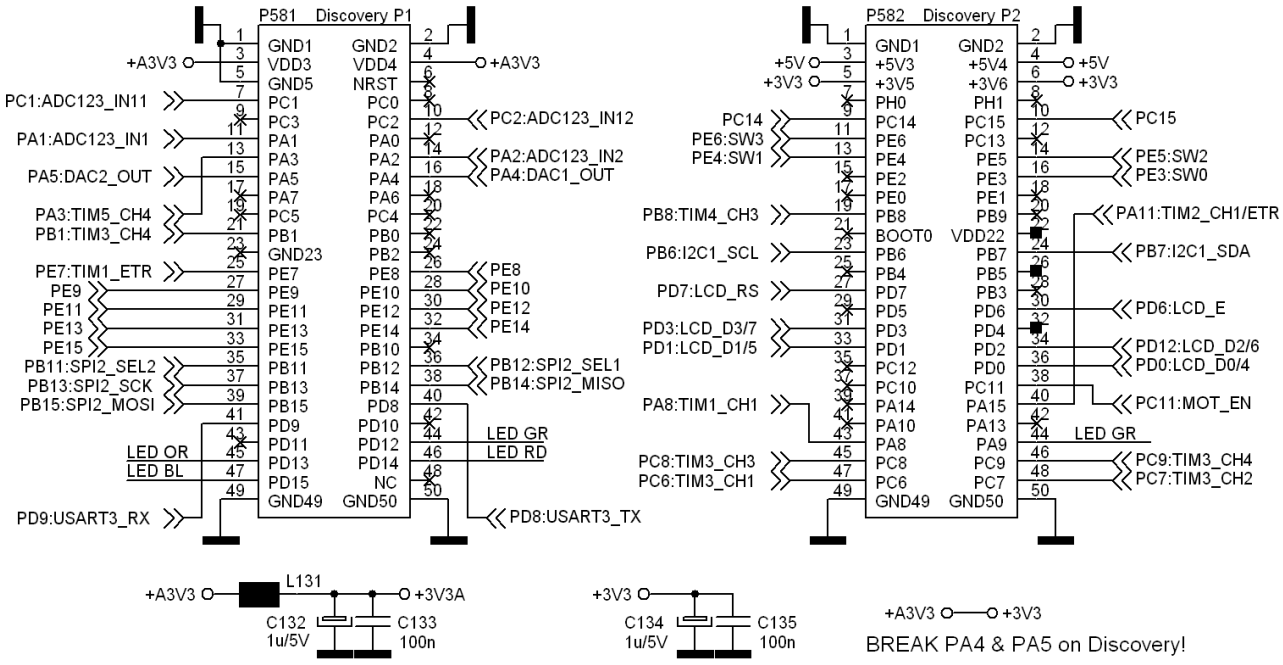


Figure 2.2.14: Signals at header connectors, STM32F4-Discovery board

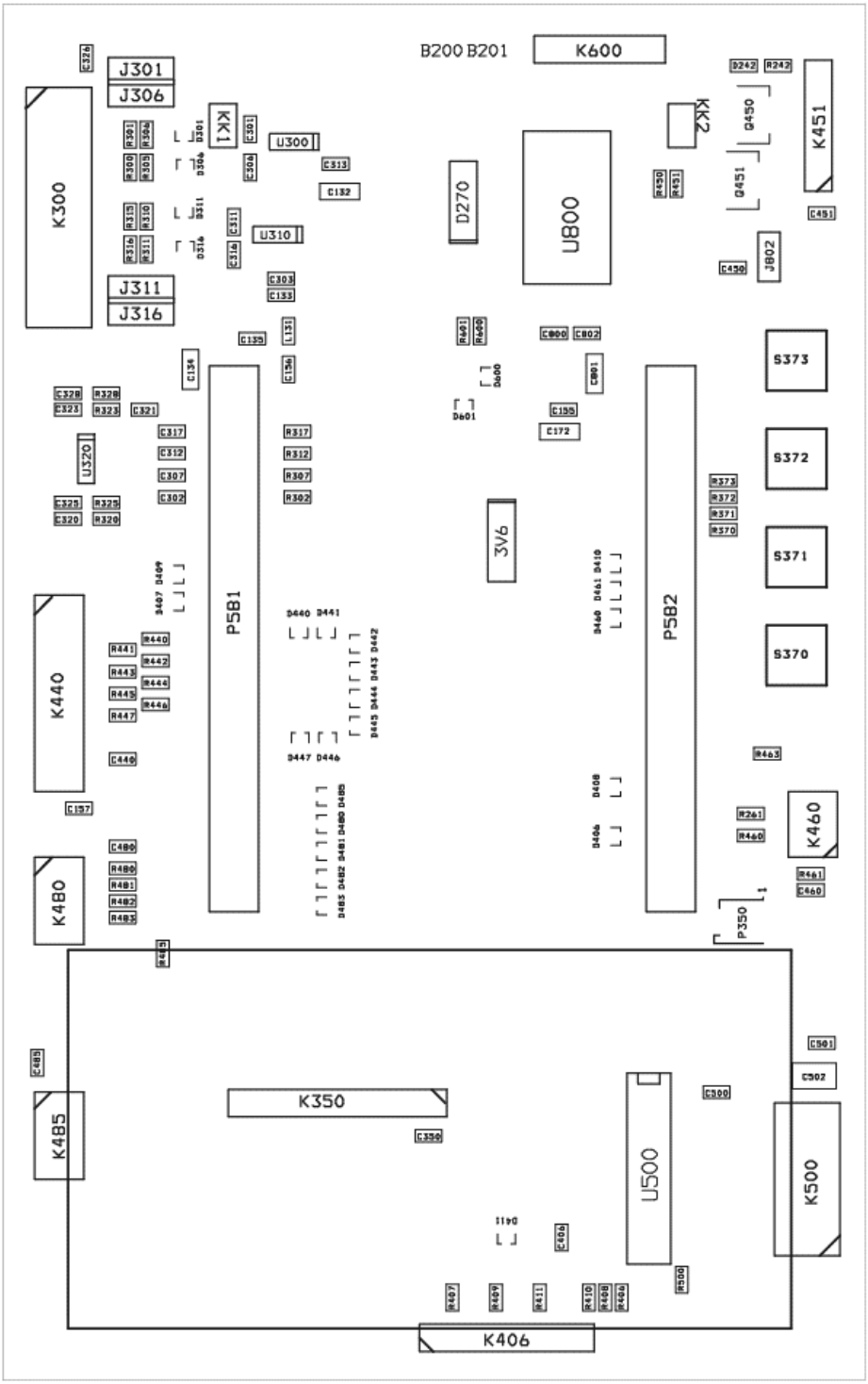
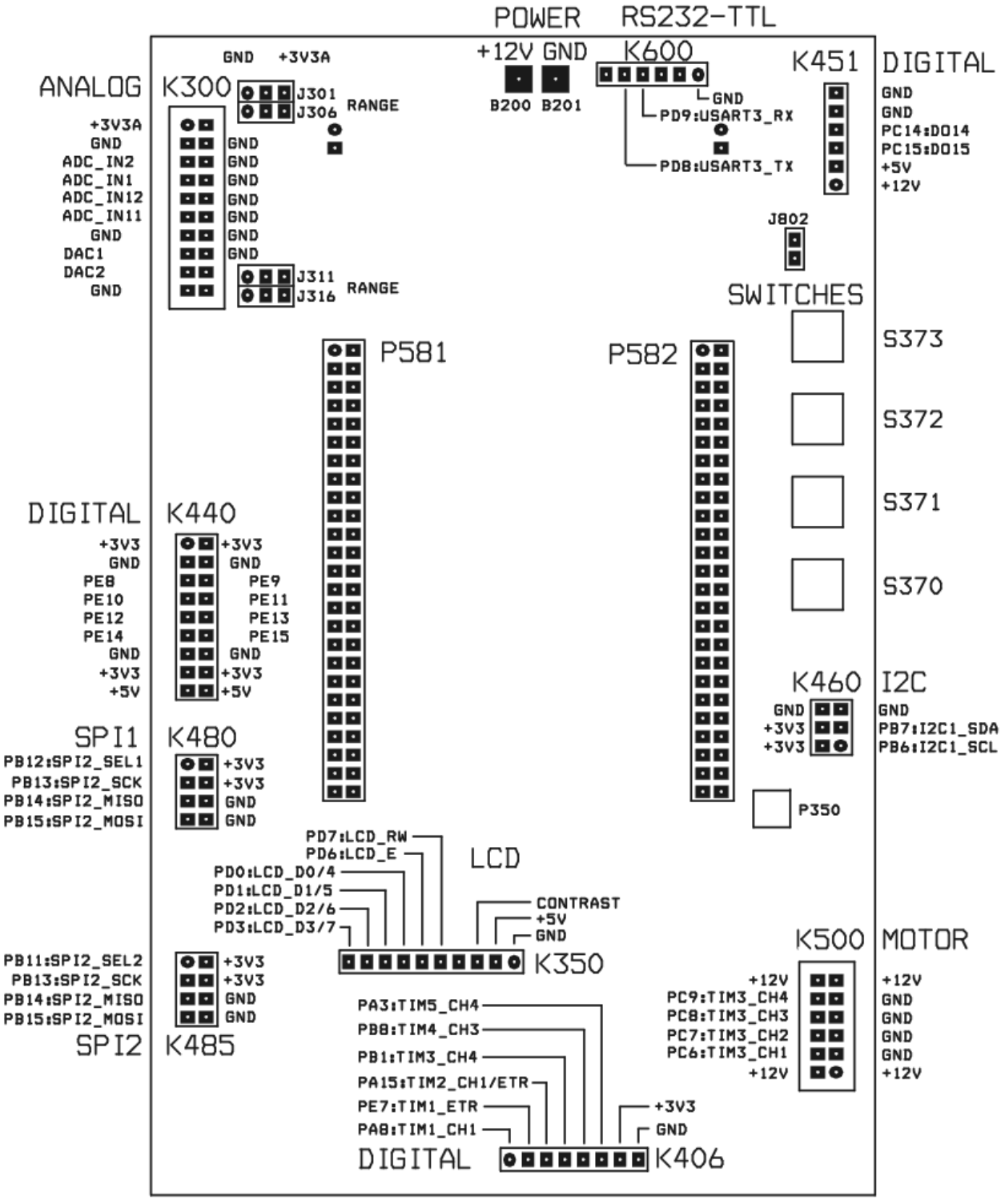


Figure 2.2.15: BaseBoard, component layout



i

Figure 2.2.16: BaseBoard, signals and connectors

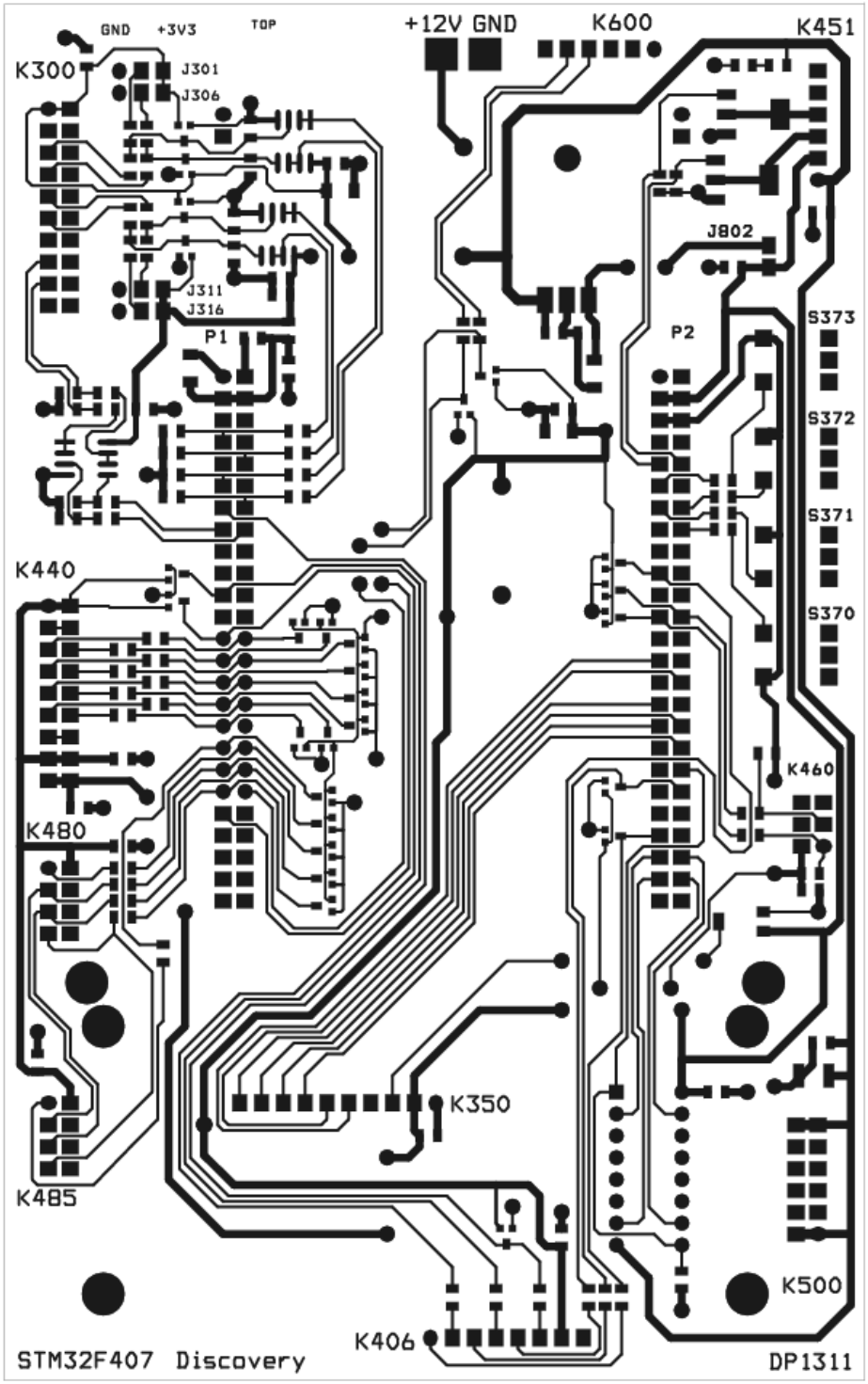


Figure 2.2.17: BaseBoard PCB, top view

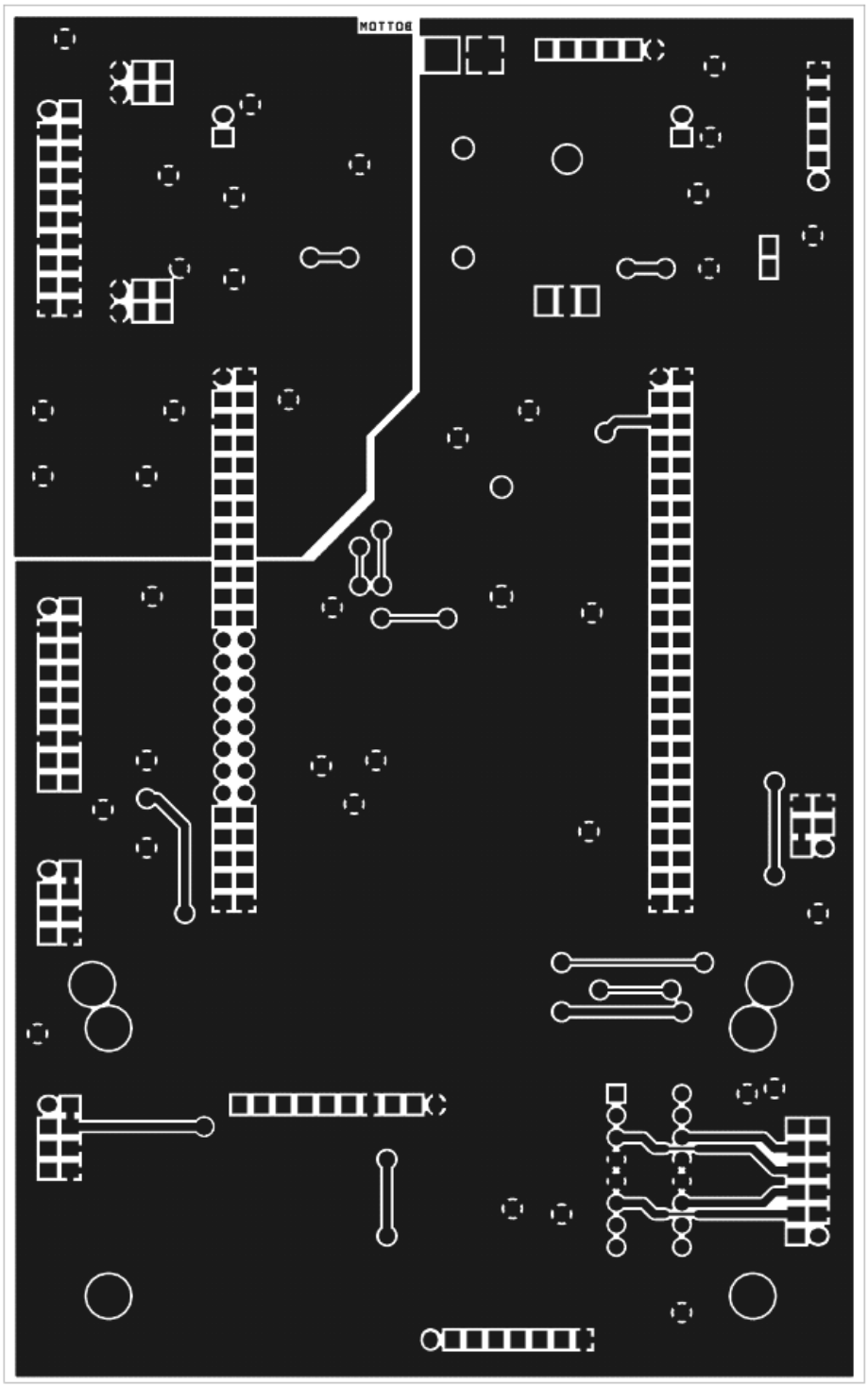


Figure 2.2.18: BaseBoard PCB, top view

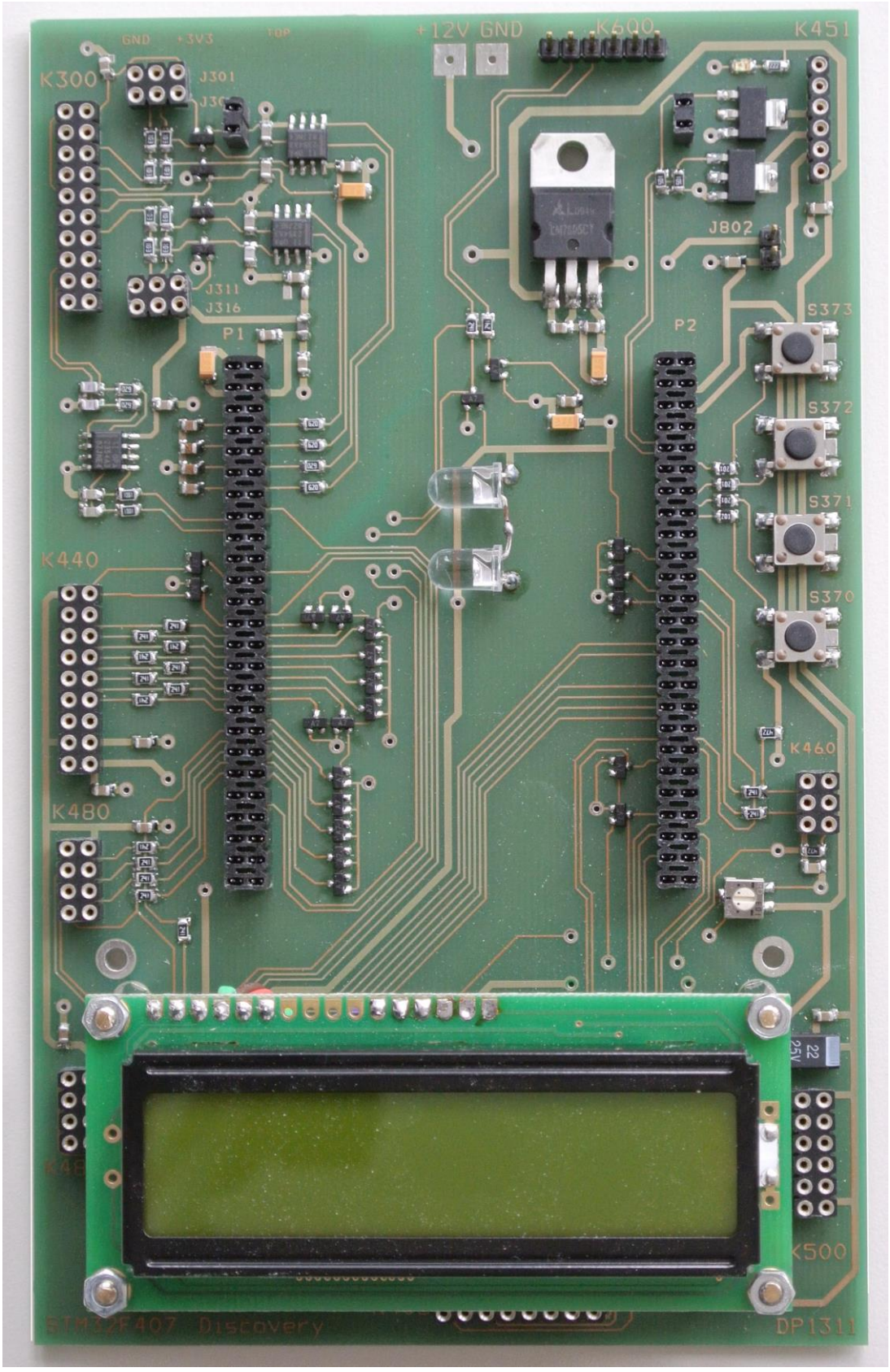


Figure 2.2.19: Photo of the finished BaseBoard without the STM32F4 Discovery board installed

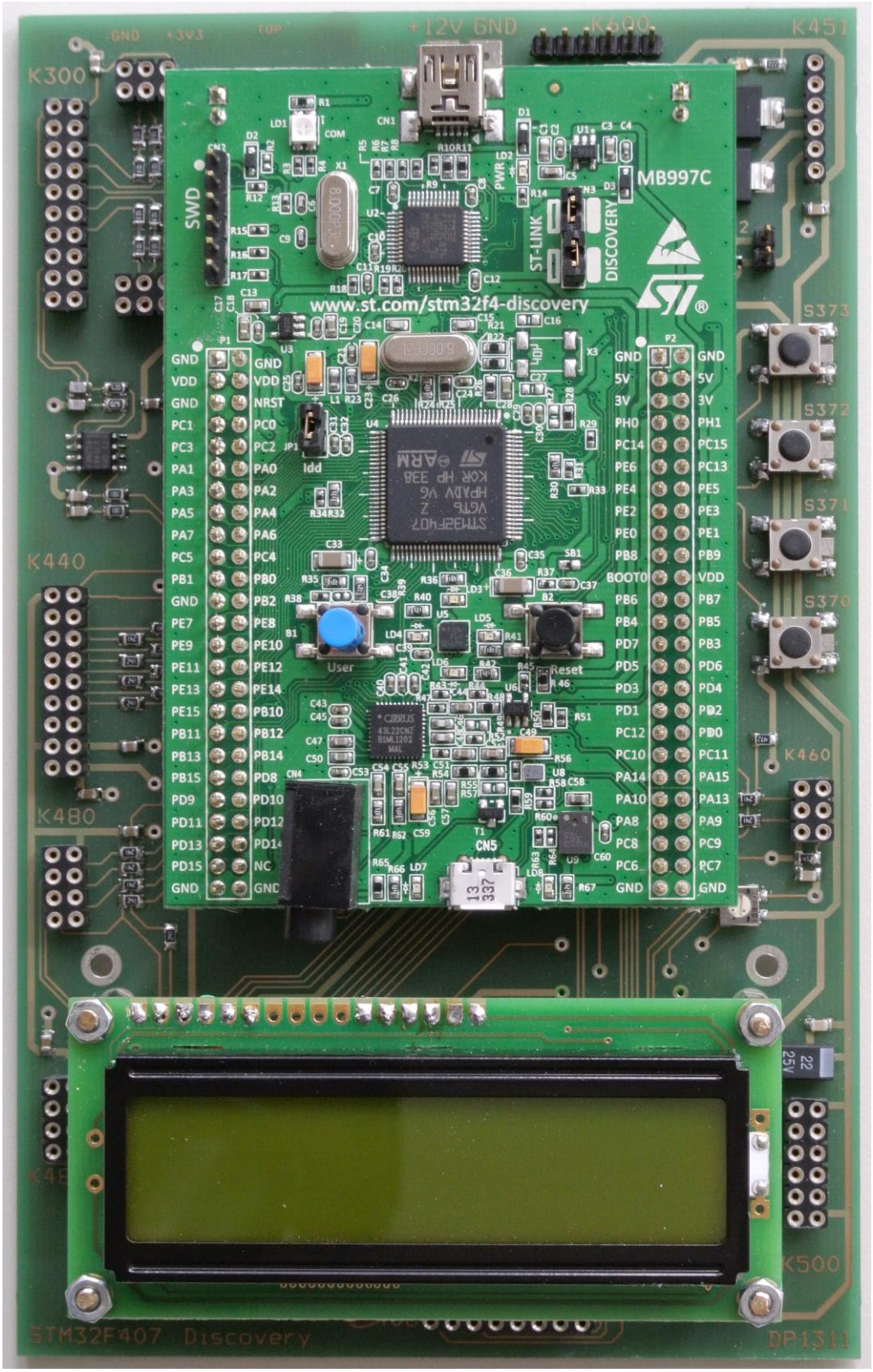


Figure 2.2.19: Photo of the finished BaseBoard with the STM32F4 Discovery board installed

3. Programming the STM32F4-Discovery

The programming environment including the settings for compiling and programming are described.

3.1. Hardware - The programming interface

A program for a microcontroller is prepared on a personal computer (PC) using a suitable set of programs. First the source code of the program is written, than this code is translated into the machine code for a particular microcontroller, the STM32F407VG in our case. The developer then loads the code into the memory of the microcontroller. Several methods for loading are known. Most universal programmers (devices that fit between a PC and the target microcontroller, and allow the transfer of the code into the microcontroller) support JTAG method. The company ST introduced a method called serial wire debug (SWD), which uses fewer wires than JTAG method. The method SWD is supported by STM32F4DISCOVERY board, and the programmer that comes between a USB connector and the SWD pins of the microcontroller is available on the Discovery board.

The STM32F4DISCOVERY board hosts additional connector for SWD signals and can also be used to program external free standing microcontroller of the same family.

3.2. Software – The IAR EWB for ARM microcontrollers

The complete procedure of writing, compiling and testing the user program can be done in a set of programs called the development suite. We are going to use the IAR (the name of the company) Electronic Work Bench (EWB) for ARM microcontrollers. The current version is 6.7. All demo programs will be prepared using the free version of the software; this version is size-limited, but our needs are way below the limits of the suite. The current version IAR EWB and instructions for the installation can be downloaded from IAR website.

The IAR EWB requires at least the following files to create the machine code for the microcontroller:

- The user program, a text file written in “C” language.
- The file “startup_stm32f4xx.s”, a text file written in assembly language; this file contains instructions for the initial set-up of the microcontroller (stack, program counter, interrupt vector table, initial system clock …; the description of the file is given in its header).
- The file “system_stm32f4xx.c”, a text file written in “C”; this file contains functions for the detailed microcontroller set-up (system clock, clock distribution …; the description of this file is given in its header).

  In our case the file may have different names; since the file contains clock configuration commands it is best to prepare several files in advance, each for a different configuration of the clock and then simply include in the process of compiling the file that corresponds to the current needs. We will run the microcontroller at its maximum speed, and the corresponding file is named “system_stm32f4xx_CLK168_HSE8.c”.

- The header file “stm32f4xx.h”; this file defines processor used, names the registers and bits inside the STM32F4xx microcontroller, and defines some register structures. This is the only file that must be included from the user program in the process of compilation. The file has been modified from the original (obtained from the ST site):
  - by uncommenting the line 68 (select the microcontroller STM32F40xx by default),
  - by uncommenting the line 88 (allow the use of standard peripheral driver) and
  - by changing the HSE frequency in line 100 (from 25000000 to 8000000, as used at the STM32F4-Discovery board).
- The header file “system_stm32f4xx.h”; allows the definition of some stuff for the use of CMSIS.
- The header file “stm32F4xx_conf.h”; defines and includes some files mandatory for the project.

The peripherals included in microcontrollers STM32F4xx are complex, and their operation is configured and monitored using several control registers for each individual peripheral, the details are given in reference manual RM0090. Bits in control registers define the operation of peripherals, and can be manipulated from the “C” language by simple writes into these registers. However, the programmer must know the exact location and function of each individual bit, which may be an overwhelming task for such a complex microcontroller. To overcome this problem a standard to ease the use of control registers has been developed. It is called ‘Cortex Microcontroller Standard Interface System’, or CMSIS for short. The standard defines a set of functions written in “C” language, each of them manipulating exactly the bits to define the operation of a peripheral. When using the CMSIS the programmer does not need to know the exact location and function of bits, he/she must only know what the peripheral is capable of and call appropriate functions, which are located in CMSIS library; functions take care of setting bits in control registers correctly. The standard covers several families of microcontrollers, and also eases porting of software from one type of microcontroller to another. The complete library for STM32F4xx series microcontrollers is available at ST site as “STM32F4xx_DSP_StdPeriph_Lib_V1.1.0” at the time of writing. The directory structure of the expanded library is given in Fig. 3.2.

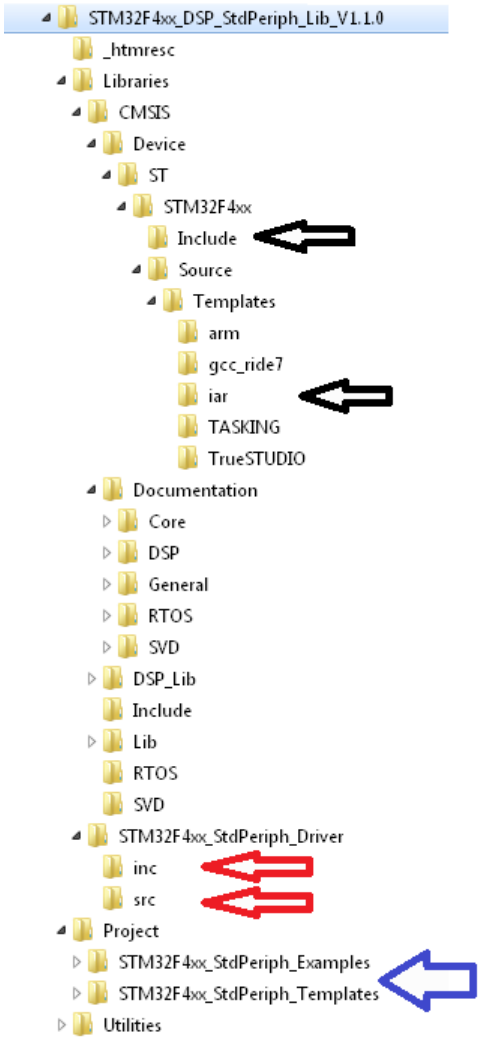


Fig. 3.2: The directory structure of the CMSIS library

The use of CMSIS standard is the reason that additional files are mandatory during the process of compilation beside the ones listed above. Two files are associated with every peripheral in the microcontroller:

- The header file defines the data structures used in accessing the peripheral and names the registers and constants (“stm32f4xx_$$$$.h”).
- The source file contains the actual functions to access the registers responsible for the behavior of peripherals (“stm32f4xx_$$$$.c”).

Here “$$$$” stands for the name of the peripheral. The compiler must know the location of all header and

include files used, and this must be specified in the compiler setup before the compilation process.

All files mentioned are available in the CMSIS library, see Fig. 3.2 for the expanded directory structure of the library. The files “system_stm32f4xx.h” and “stm32f4xx.h” are located in directory “Include”, upper black arrow. The programming environment dependent file “startup_stm32f40x.s” is located in directory “IAR”, lower black arrow. Please note that these files are original, unmodified ones, and it is most convenient to store the modified versions to a separate directory and call them from there. CMSIS header (*.h) and source (*.c) files describing peripherals are located in directories “inc” and “src” respectively, both red arrows. Some examples on the use of CMSIS library are given in directory “STM32F4xx_StdPeriph_Examples”, and templates for different compilers are given in “STM32F4xx_StdPeriph_Templates”, blue arrow.

The process of downloading the compiled program into the microcontroller requires one additional file with the description of downloading procedure and the destination memory for the program, this file is called linker configuration file. There are three options to choose from as far as the destination memory is concerned: the file may be programmed into the flash memory, the file may be downloaded into the random access memory inside the microcontroller or the program may be loaded into an external RAM. All three configuration files have the extension “.icf”, and can be found in “STM32F4xx_StdPeriph_templates\EWARM”. Since we want to make programming permanent we will only use the file named “stm32f4xx_flash.icf”.

Two additional files were prepared for these experiments, “LCD2x16.c” and “dd.h”. The first contains functions to display character strings and numbers at the alphanumerical LCD display, the second one defines some constants and interfacing pins. Both files should be made available during the compilation process.

3.4. Creating of a new workspace and a project within

Programs for microcontroller are prepared in a workspace, an imaginary working environment containing all the user supplied stuff to compose the final machine code. The workspace is associated with a folder on a computer disk. The workspace contains projects; each project holds a complete set of files (or references to files common to several projects) that constitute one user program. Typically all files needed for a project are stored in a designated sub-folder within a folder for the workspace to avoid the naming confusion of individual files within the workspace.

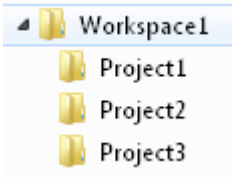


Figure 3.3: Folders created to host the workspace and three projects

The procedure for creating a new workspace and a project within will be described step-by-step. The directory structure to hold the workspace and projects is created first using the Windows Explorer,

Name	Date modified	Type	Size
dd	2.12.2013 13:57	H File	2 KB
LCD2x16	3.12.2013 8:50	C File	5 KB
startup_stm32f40xx	10.1.2013 11:54	S File	25 KB
stm32f4xx	17.12.2013 9:17	H File	533 KB
stm32f4xx_conf	5.12.2013 9:15	H File	4 KB
stm32f4xx_flash.icf	28.3.2013 9:09	ICF File	2 KB
system_stm32f4xx	10.1.2013 11:54	H File	3 KB
system_stm32f4xx_CLK168_HSE8	4.4.2013 11:03	C File	22 KB

Figure 3.4: The mandatory files are copied into the project directory

the directory structure is given in Fig. 3.3, for hosting three projects (Project1, Project2 and Project3) within a workspace called Workspace1. Additional directories can be added later to host more projects in the same workspace. A set of files as listed in Fig. 3.4. is copied into each of the project directories for easy access and inclusion into the user program. Note that some of these files are modified versions of the originals given in the CMSIS library.

3.4.1. Create a workspace

Initially the "IAR Embedded Workbench" is opened resulting in an empty space in the center of the program window and empty workspace in the left part of the window, Fig. 3.5. This state can be achieved also by clicking the "File" option from the menu, followed by "New" and "Workspace".

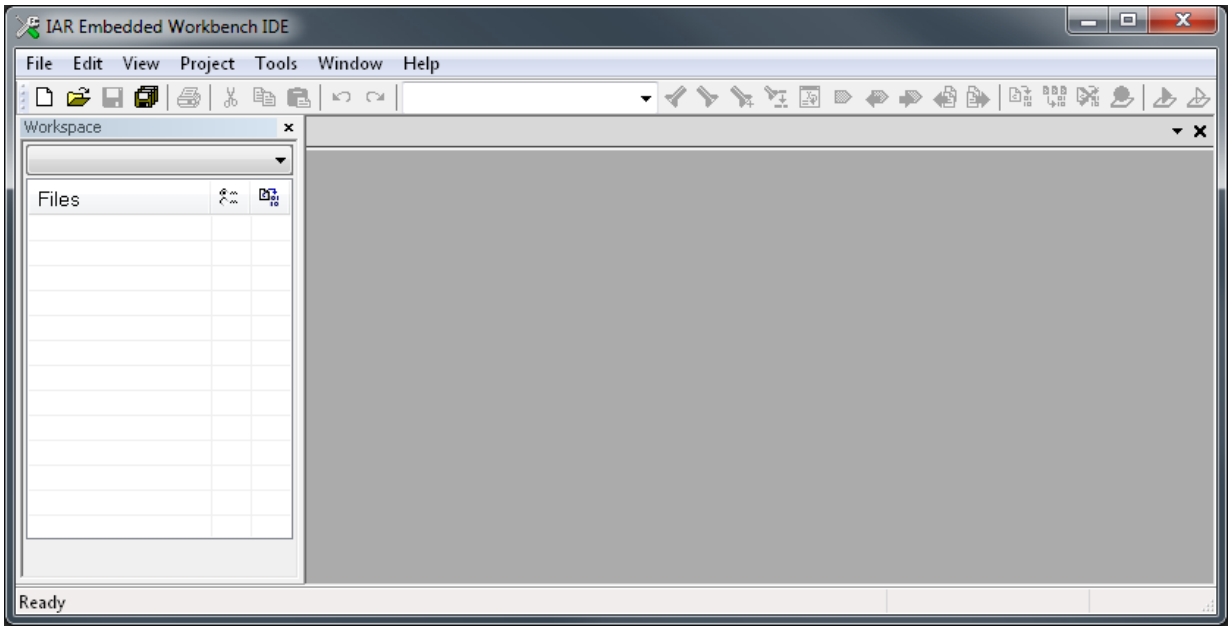


Figure 3.5: A newly opened »IAR Embedded Workbench« with an empty workspace on the left

3.4.2. Create a project

- Click "Project" option in the menu, and then click "Create New Project …" to get a window as shown in Fig. 3.6.

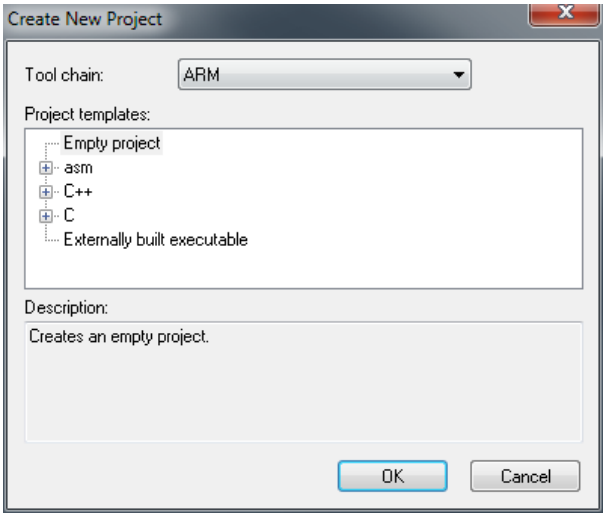


Figure 3.6: A window to create a new project

- Select “Empty project” and click “OK” to get a window “Save As”. Navigate to the folder where the project is to be saved (The “Workspace1\Project1” in our case). Under “File name:” type in the name of new project, like “MyFirstProject”, and click “OK”. The skeleton of the new project will be created in the stated directory, and the Workspace window (Fig. 3.5, left part of the window) of IAW EWB will contain the name of the newly created project.
- Save the newly created workspace by clicking “File” option in the menu and then clicking “Save Workspace”, navigate to the appropriate folder for the workspace (“Workspace1” in our case), state the name (“MyFirstWorkspace” for instance) of the newly created workspace in “File name:” and confirm saving by clicking “Save”.

3.4.3. Add files to project

Every project must contain at least files listed in the Fig. 3.4. Some of them are included from the user program, some are specified at the time of compilation, and some must be added in the project itself. The last are “startup_stm32f4xx.s”, “system_stm32f4xx_CLK168_HSE8.c” and the “C” file containing the user program.

- The first two files are added to a project by right-clicking the name of the project in the workspace window and selecting option “Add” -> “Add Files” from a drop-down menu. This opens a window where appropriate files can be selected (from the “Workspace1\Project1” folder in our case).
- Additionally, a new file with the user program must be created. A new file is created by clicking the leftmost icon in the toolbar (a symbol for empty pages). An empty page appears in the right IAR window, and the user can write new text onto the page. The user program must start with the line “#include “stm32f4xx.h”” to allow the use of predefined names for register in the microcontroller. The file must include at least the function “void main (void)”. The file should be saved in the folder of the choice (“Workspace1\Project1” in our case) and can have any name, like “MyFirstFile.c”. This file must also be added to the project using the same procedure as stated previously.

The result is the IAR EWB window shown in Fig. 3.7.

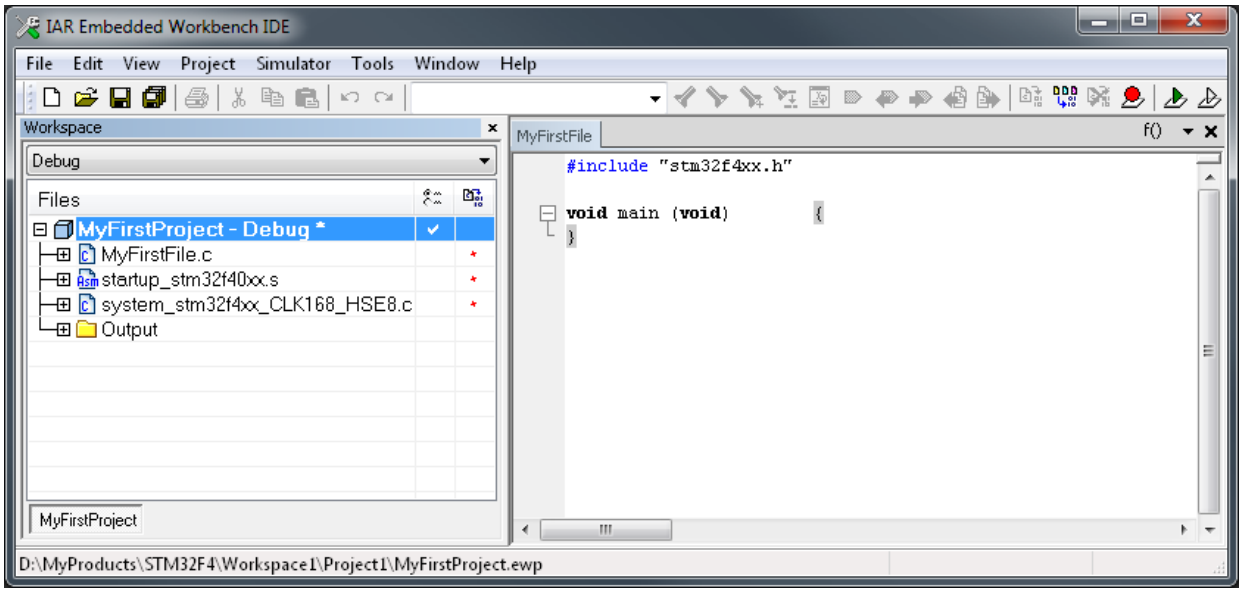


Figure 3.7: A fully defined minimal project within a workspace

3.4.4. **Define options for a project**

The process of altering the user provided source code into machine code consists of compiling and linking. The IAR Embedded Workbench does both, but needs additional information for the process. This information can be supplied by clicking the option “Project” in the menu and then “Options..”. This brings out a window where options for the operation of the compiler and linker can be configured.

- The IAR Embedded Workbench needs to know the microcontroller to compile for. Under “General Options” click “Device” and select the appropriate ST microcontroller (STM32F407VG) as shown in Fig. 3.8.

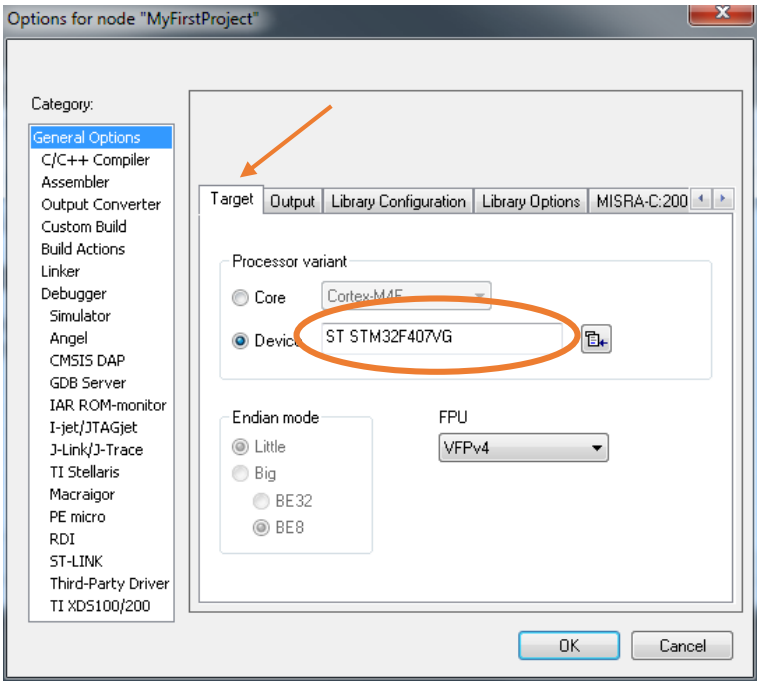


Figure 3.8: Select the microcontroller

- Enable the CMSIS standard and libraries by ticking the appropriate tick box, Fig. 3.9.

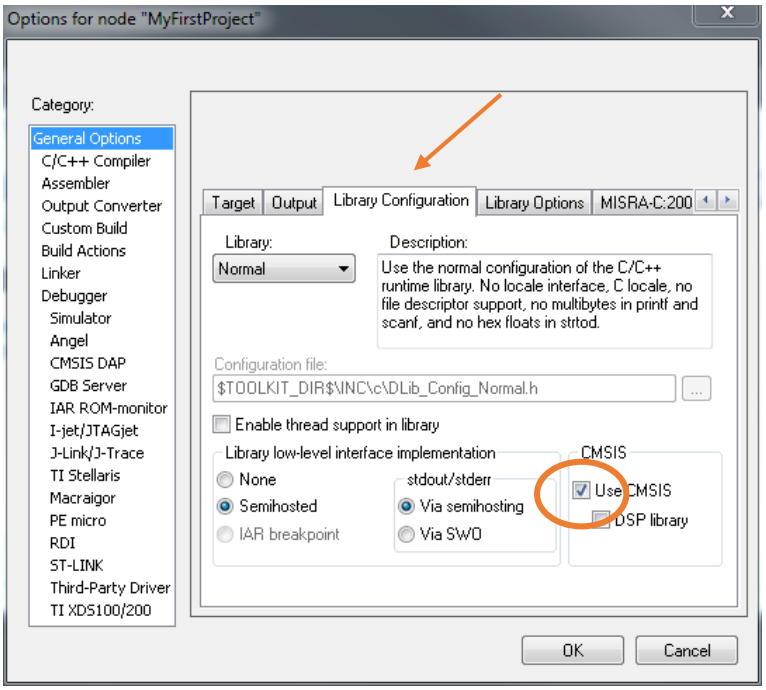


Figure 3.9: Enable the use of CMSIS standard

- Set the level of optimization for compiler, Fig. 3.10. The default is “Low”, but we can safely select “Medium” to speed-up the execution of the program. Selecting the option “High” enables the compiler to actually modify the user program in order to speed-up the execution, and is to be used with caution.

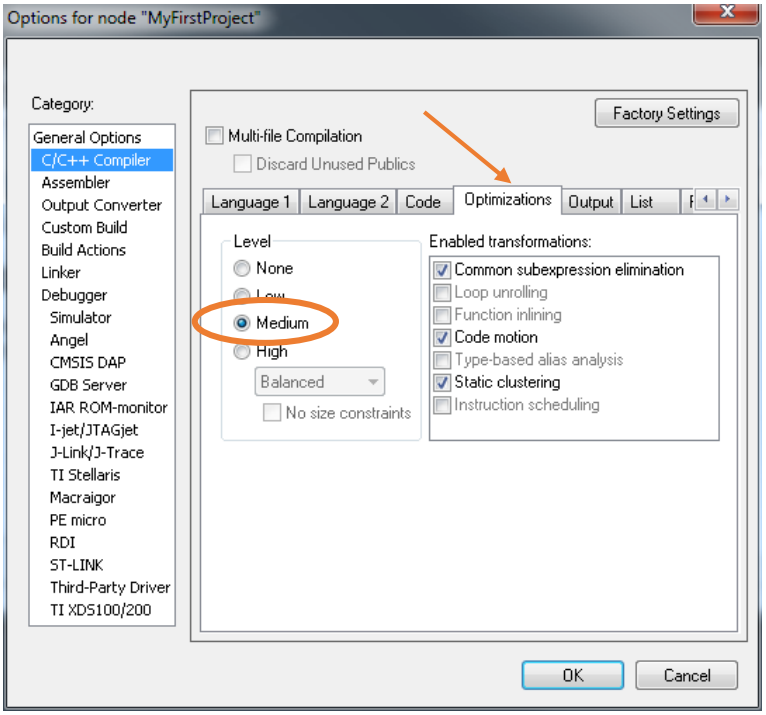


Figure 3.10: Enable optimization, level “Medium”

- The IAR Embedded Workbench needs to know the location of header and include files conforming to CMSIS standard to handle the peripherals, Fig. 3.11. The folders for these files are specified relatively to the directory where the project is located (variable $PROJ_DIR$). From this directory

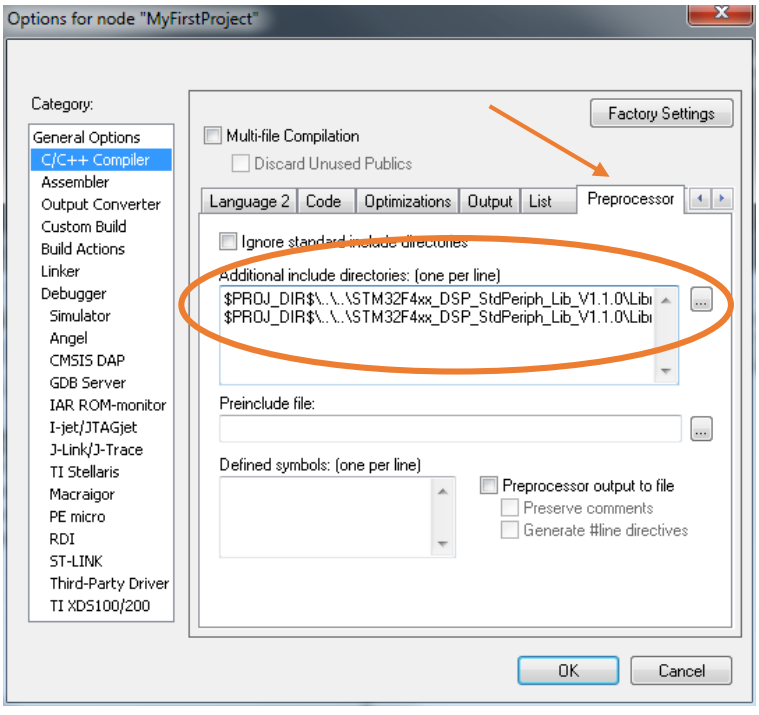


Figure 3.11: State the location of CMSIS header and include files

the relative path leads two levels up ("\..\..") and then downwards to directory named "\STM32F4xx_DSP_StdPeriph_Lib_V1.1.0\Libraries\STM32F4xx_StdPeriph_Driver\inc" and "\STM32F4xx_DSP_StdPeriph_Lib_V1.1.0\Libraries\STM32F4xx_StdPeriph_Driver\src", see Fig. 3.2 for reference.

- The machine code can be prepared to run from flash memory, from RAM or from external RAM, but the linker must know the desired destination for the code. We define if by selecting the appropriate linker configuration file "stm32f4xx_flash.icf" as shown in Fig. 3.12.

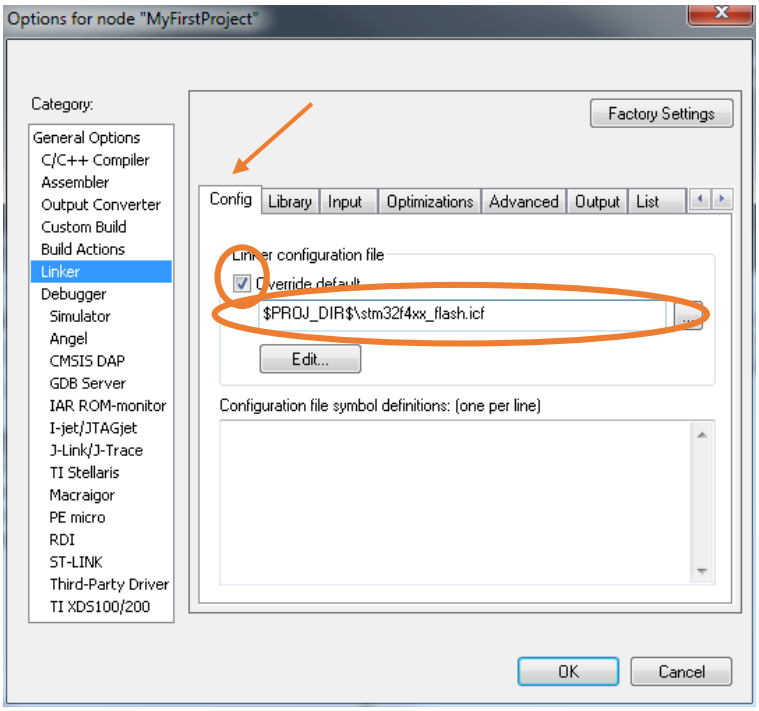


Figure 3.12: The machine code will be downloaded into the flash memory, and it must be linked accordingly

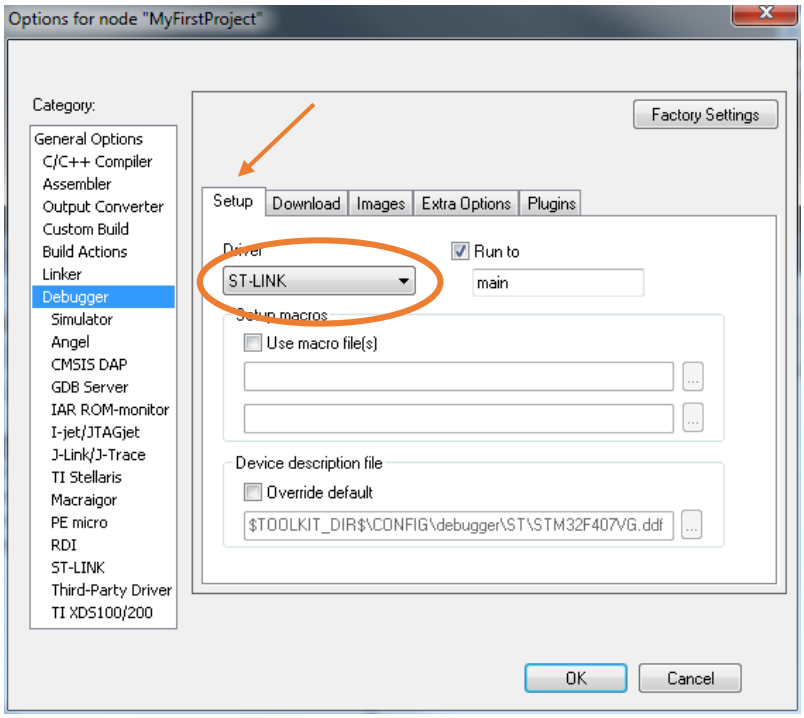


Figure 3.13: ST-LINK / 2 programmer is used for downloading and debugging

- We will use the ST-LINK / 2 programmer embedded onto the STM32F4-Discovery board to download and debug the machine code, so we need to tell the IAR EWB about our selection, see Fig. 3.13.
- Since we want to load the machine code into the flash memory the programmer must use the appropriate protocol for the download, namely the flash loader, see Fig. 3.14.

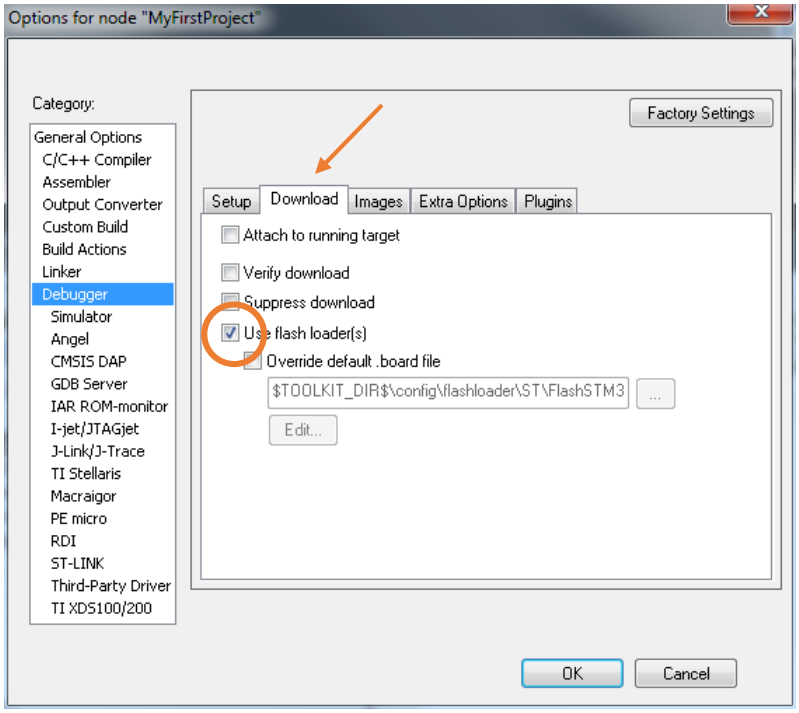


Figure 3.14: The programmer should load the machine code into flash memory

- The target microcontroller is connected using a ST-LINK and SWD protocol, Fig. 3.15.

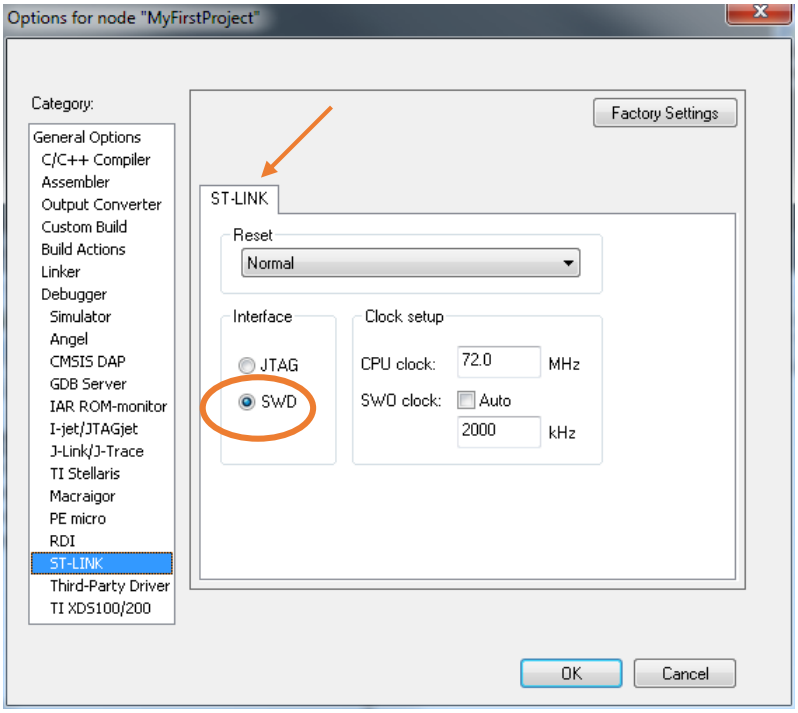


Figure 3.15: Use SWD as the preferred protocol for downloading the code

3.5. Compiling and debugging a project

Once the user program is typed-in and saved, and all the above settings are in place, one can run the compiler and linker to convert the user program into the machine code.

- The compilation and linking can be initiated either from the menu ("Project" -> "Compile") or simply by pressing F7. There is also an icon on the toolbar to initiate the compilation (Fig. 3.7, fifth icon from the right on the toolbar).
- The compilation, linking and download to the target processor can be initiated by simply clicking the second icon on the right from the toolbar. If either compiler or linker finds errors during the process, they issue warning and/or error messages and stop the process.

The debugging of the program can be either pure simulation in software of the personal computer or running in real hardware.

- When the first is desired, the step described in Fig. 3.13 must be different: the "Simulator" must be selected instead of "ST-LINK". The machine code will not be sent to the microcontroller, but kept inside the personal computer instead. The personal computer will read the machine code step by step and simulate the behavior of real microcontroller. This mode is essential for testing algorithms for all possible values of input variables, but cannot simulate the real hardware (interrupt requests, timers/counters, ADCs, DACs, …).
- When the second is desired, the real hardware with the microcontroller is mandatory. In this case the program and hardware can be fully tested in real life situation, as we are going to do.

The download of the program is initiated by pressing the button with the green triangle pointing right, or can be initiated using the menu. The suite then enters the debug mode, and Fig. 3.16 gives a sample of the debug session in progress.

The execution of the program is started by clicking the "Run" button, the one with three arrows to the right, just left to the button with the red "X". Clicking the button with a hand stops the execution, and clicking the button with a single arrow pointing to the left resets the microcontroller. Various options for single-stepping are available, the currently executed line is marked with a green arrow left of the listing. The simulation/emulation stops when the user clicks the button with the red "X".

The execution of the running program can be stopped at a certain line of the program using breakpoints. A breakpoint can be set onto a specific line of code using the double click in front of the line, which brings a red dot beside the line with the breakpoint. Please note that the compiler optimizes the user supplied source code, and some lines of the user program can be omitted and embedded in nearby lines. Breakpoints cannot be placed onto the missing lines, and in such case the suite places a breakpoint to the nearest existing line of the user program.

The content of all registers is available for inspection and alteration when the program is stopped either manually or after a breakpoint. One needs to click option "View" in the main menu and then "Registers" in the drop-down menu. This opens a new window inside the IAR EWB window with the content of the CPU registers, see Fig. 3.16, right. The user can select to inspect or modify registers belonging to peripherals by clicking the drop-down box at the top of this window and selecting the desired peripheral. Initially, the content of registers is shown in hex notation, but can be broken down to bits by clicking the + sign in front of the register name.

The value of variables used in program can be shown by clicking the "Watch" option in the "View" menu. This opens a new window, and the user can write the name of the desired variable into an

empty slot. The value of the variable is returned to the right of the name when the program is stopped. It is shown in hex notation initially, but can be converted to decimal, ASCII or binary on demand. Due to optimization during the compiling of the program some variables might not be available for watching. In such case the value will not be returned, but a message will be displayed instead. Variables can be also inspected or modified when “View” -> “Quick View” is selected, but with different options.

Figure 3.16 gives an example of the IAR Embedded Workbench in the debug mode showing the project window (where green arrow is used to point to the line to be executed when simulation is started) and register window (showing the content of register associated with port A in hex notation.

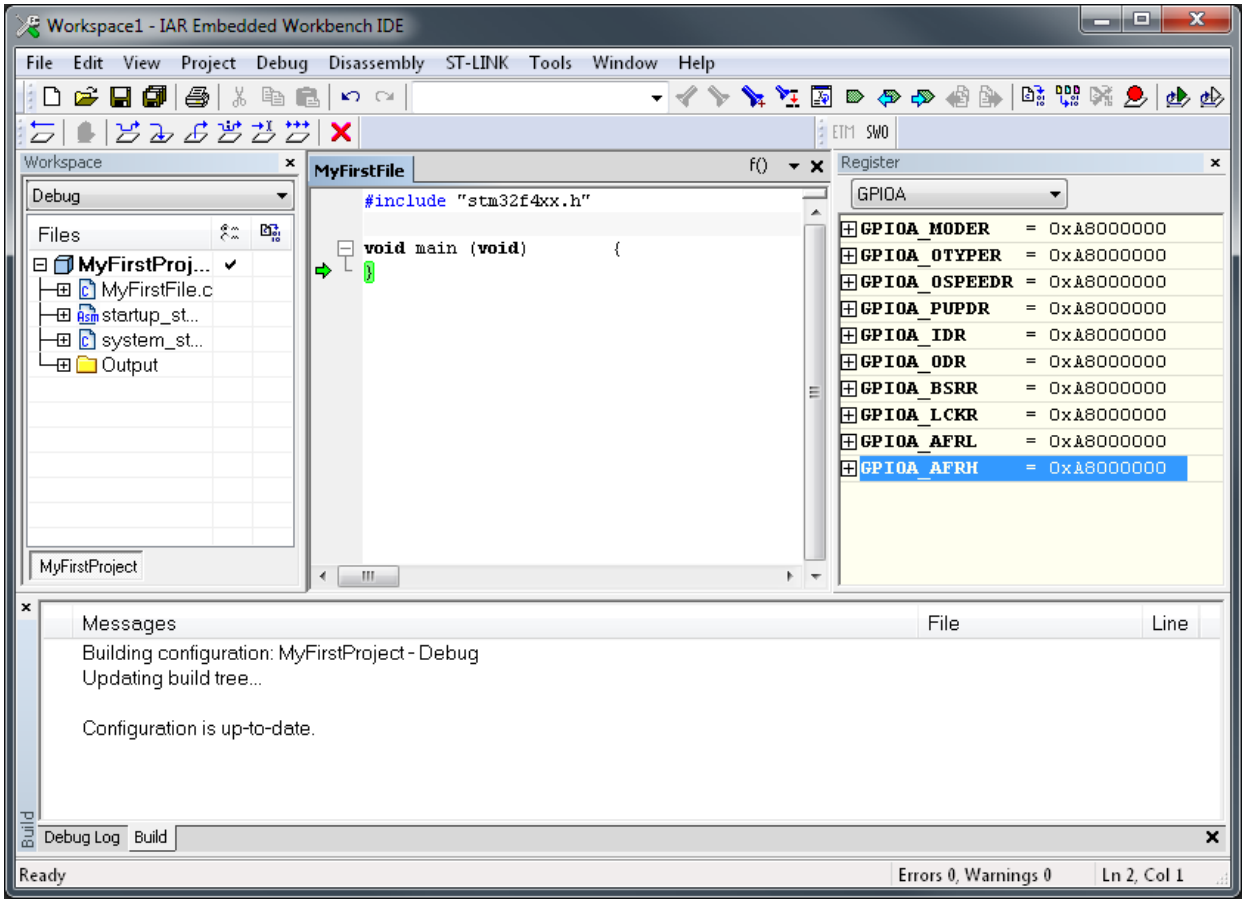


Figure 3.16: An example of a debug session in progress

4. The use of ports

The configuration and the use of microcontroller ports is given.

4.1. Hardware – a port

The microcontroller STM32F407VG has five sets of pins (16 pins per set), each set is called a port. Each pin of every port can be individually configured as input or output for a digital signal, and for each of these options additional settings to accommodate the desired type of a signal are available. Alternatively, each pin can handle one from a set of 16 predefined alternate functions. Both settings and functions can be configured from within the user program; writing the appropriate values into registers inside of microcontroller defines the properties of pins and ports.

A basic block diagram of a pin and associated circuit is given in Fig. 4.1. The pin itself is shown on the right side of the figure (I/O pin). Two diodes are connected to ground and power supply respectively to protect (to some degree) the microcontroller from overvoltage connected by the unwary user. The microcontroller can write to a pin by writing the data into the 'Output data register' shown in the left part of the figure. This register is 16 bits wide, and each bit is connected to one pin of the port. The microcontroller can read the pin by reading the content of the Input data register, also shown in the left part of the figure. The register is again 16 bits wide, and each bit is linked to one of the pins of the port. The intermediate hardware between pin and data registers is shown in two dashed rectangles named 'Input driver' and 'Output driver'. The properties of these drivers are programmable.

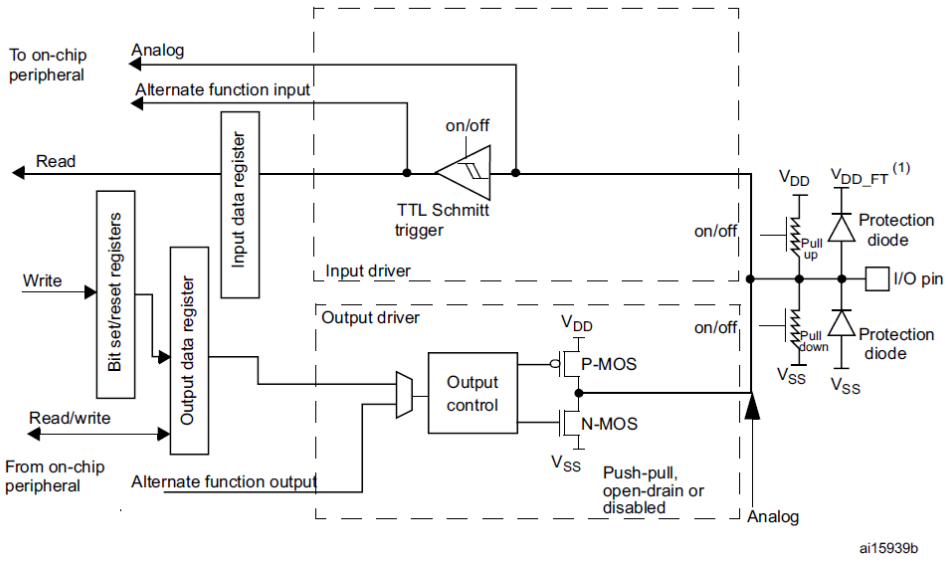


Figure 4.1: A basic block diagram of the circuit associated with a pin, RM0090, pg. 266

A pin can be used as an output; for instance when the microcontroller sends signal through the pin to a LED. In this case both or at least one of the two MOS transistor constituting 'Output driver' must

be active, and this mode of operation must be known in advance. Alternatively, a pin can used as an input; for instance when accepting the value from a switch. In this case both MOS transistors must be disabled in advance not to interfere with the signal from the switch. The direction of the signal flow is configured during the programming.

When pin is configured as an input, the default level at the input is not defined, i.e. the pin is floating. This might not be convenient. Consider the situation where a pushbutton is connected between a pin and a power supply. The signal at the pin is defined when the pushbutton is pressed, but left floating otherwise. This can be fixed by connecting a resistor from the pin to ground; this defines the signal when the pushbutton is not pressed. Two resistors to terminate the pin to either ground or power supply are available for every pin, and can be activated by configuration.

The output driver can work at different speeds. The speed is related with the current consumption: faster action requires more current. The speed can also be configured.

In addition to simple passing of signals to and from the microcontroller pins can be used for so called alternate functions. This includes the passing of analog signals to ADCs and DACs, and passing of digital signals to and from timers for instance. In all alternate functions cases input and output drivers must be adjusted to the task. Again, this ought to be configured prior to the use.

The reference manual RM0090 gives a detailed description of ports and settings available in chapter 8. There is a table with all registers associated with the properties of ports, and meaning of all bits within these registers. The properties of a port can be configured by writing in these registers. However, a detailed knowledge on individual bits of within the registers in mandatory, and this can be overwhelming task. The CMSIS standard provides a convenient set of functions and keywords to ease the setting-up of the ports.

The operation of any peripheral or the CPU is timed by a designated clock signal. In order to conserve power most of the clock signals for peripherals are disabled on reset, and must be enabled before the peripheral can be used.

4.2. Software – the CMSIS header and include files

The CMSIS functions to define the operation of the port are given in “stm32f4xx_gpio.c” (GPIO == ‘General Purpose Input Output’), while the CMSIS functions to enable the clock signal for individual peripherals are available in “stm32f4xx_rcc.c” (RCC == ‘Reset and Clock Control’). Note the name of these files, especially the part after the ‘_’; it reflects the peripheral referred by the file. Both files must be included from the user program to use functions from them. The keywords and definitions used by these functions are stored in header files “stm32f4xx_gpio.h” and “stm32f4xx_rcc.h” respectively. These two files are automatically included in the compilation process from the header file “stm32f4xx.h”, which must be included from within user program.

A user program to manipulate any of the ports using the CMSIS functions must therefore start with three “#include” commands:

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"
```

The first of the files to include is modified from the original in CMSIS library, and is stored locally in the project directory. Other two files are originals from the CMSIS library, and pointers are set to them in “Options…” before the compilation, see chapter 3 for details.

A brief instruction on the use of CMSIS functions for manipulating a port is given at the beginning of the file “stm32f4xx_gpio.c” in section “How to use this driver”. The recommended procedure is listed in steps marked with ‘(#)’:

- enable the clock for the port using function “RCC_AHB1PeriphClockCmd”,
- configure the port (possible configurations are listed) using the function “GPIO_Init”,
- configure alternate functions if applicable, and
- use the port.

Similar sections with brief instructions can be found in all “stm32f4xx_$$$$.c” files containing functions for other peripherals.

A simple example follows. The microcontroller is used as a counter; it increments a variable, and the value of this variable is sent to port E. The STM32F4-Discovery board has most of microprocessor pins accessible, but many of them are best used for special purposes, like ADCs, DAC, counters, communication, etc. The BaseBoard exposes port E, pins 8 to 16, at connector K440 as general purpose IO pins, and these will be used to pass the content of the counter. Due to the selection of pins used it is best to increment the counter by LSB of the exposed bits, therefore by 0x100. The listing of the core of the program is given below:

```
while (1) {                          // about 50ns per iteration
    number += 0x100;                 // increment at BIT_8
    GPIOE->ODR = number;             // send to PortE
  };
```

An infinite ‘while’ loop is used, and within the loop the content of the variable ‘number’ is first incremented by 0x100 (100_{HEX} == 256_{10}, increment by one at bit 8), and then sent to the port E. The name ‘GPIOE->ODR’ points to port E, output data register. Alternatively one could use here the CMSIS function “GPIO_Write” to do the same (CMSIS function performs syntax checking), but the direct writing into the output data register used here is faster.

As stated in brief instructions above the user must enable the clock for the port E and configure the port before the use. Both actions are best enclosed into a function named “GPIOEinit” that is called prior to the use of the port:

```
void GPIOEinit (void)   {
GPIO_InitTypeDef  GPIO_InitStructure;                                                  // 2

  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE);                                // 4

  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_8  | GPIO_Pin_9  | GPIO_Pin_10 | GPIO_Pin_11;       // 6
  GPIO_InitStructure.GPIO_Pin  |= GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;       // 7
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_OUT;                                       // 8
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;                                       // 9
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;                                   // 10
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;                                    // 11
  GPIO_Init(GPIOE, &GPIO_InitStructure);                                               // 12
}
```

The syntax of statements follows the CMSIS standard.

- The first action is performed by the statement in the fourth line of the listing above where the function “RCC_AHB1PeriphClockCmd” with two arguments is called. The first argument is the name of the port to manipulate the clock for (‘RCC_AHB1Periph_GPIOE’), and the second is the actual action (‘ENABLE’ or ‘DISABLE’). Please note that the last letter of the first argument defines the port to be one from A to E.
- The second action is performed by a call to a function “GPIO_Init” in the last line of the function. This function requires two arguments, the first one being the name of the port, GPIOE in our case; please note that the last letter of this argument defines the port to manipulate. The second argument is a pointer to the data structure named ‘GPIO_InitStructure’ containing the desired settings. This data structure is declared in the second line of the listing above, and then

initialized in lines 6 to 11. The structure has five members as defined in the "stm23f4xx_gpio.h", line 124. Each member can have different values as defined in the same file, lines 64 to 168. The names used tend to be human friendly and understandable. For reference:

- Lines 6 and 7 in the listing define the pins to be affected by initializing the member called '.GPIO_Pin'. In this example eight pins are used, and their names are or-ed together in two consecutive lines of the function. The names of pins are defined as hex values in "stm23f4xx_gpio.h", lines 151 to 168, and or-ing them builds a hex number that can be directly written into a designated register of the microprocessor.
- Line 8 of the listing defines the direction of the signal flow by initializing member called '.Mode' for pins selected by previous member of the structure. The options are input, output, alternate function or analog, and are declared in "stm23f4xx_gpio.h", lines 64 to 71. The user-friendly version reads as 'GPIO_Mode_IN', 'GPIO_Mode_OUT', 'GPIO_Mode_AF', and 'GPIO_Mode_AN' respectively.
- Line 9 defines the use of both MOS transistors in output driver by initializing the member '.GPIO_OType'. The options are: 'GPIO_OType_PP' (push-pull configuration, both transistors are used) and 'GPIO_OType_OD' (open-drain, only the lower transistor is used).
- Line 10 defines the speed of the selected pins by initializing the member '.GPIO_Speed'. The options are 'GPIO_Speed_2MHz', 'GPIO_Speed_25MHz', 'GPIO_Speed_50MHz', and 'GPIO_Speed_100MHz', as defined at lines 88 to 96 of "stm23f4xx_gpio.h".
- Line 11 defines the use of termination resistors by initializing the member '.GPIO_PuPd'. Valid options are 'GPIO_PuPd_NOPULL' (resistors not used), 'GPIO_PuPd_UP' (pull-up resistor used), and 'GPIO_PuPd_DOWN' (pull-down resistor used), as defined in lines 101 to 106 of the "stm23f4xx_gpio.h".

Similar use of header and source files can be traced throughout configuration of all peripherals using the CMSIS standard. Not all members of structure need to be initialized, since for certain combinations some settings are not possible; this will be demonstrated soon. A source file named "stm32f4xx_$$$$.c" contains brief instructions on the use of functions and functions to manipulate the peripheral themselves, and the header file "stm32f4xx_$$$$.h" defines the required data structures and options.

The complete listing of the program to increment the counter and send its content to port E, bits 8 to 16, is given below.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"

void GPIOEinit (void)   {
GPIO_InitTypeDef  GPIO_InitStructure;

  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE);

  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_8  | GPIO_Pin_9  | GPIO_Pin_10 | GPIO_Pin_11;
  GPIO_InitStructure.GPIO_Pin  |= GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_OUT;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;
  GPIO_Init(GPIOE, &GPIO_InitStructure);
}
```

```
void main (void)  {
unsigned int number = 0;

  GPIOEinit();

  while (1) {                   // about 50ns per iteration
    number += 0x100;            // increment at BIT_8
    GPIOE->ODR = number;        // send to PortE
  };
}
```

The next example demonstrates the use of pushbuttons mounted onto the BaseBoard and LEDs available on the STM32F4-Discovery board. The program periodically reads the state of pushbuttons, and turns the LED that corresponds to the pushbutton ON when the pushbutton is pressed.

The pushbuttons are named S370 to S373, as shown in chapter 2, Fig. 2.2.2. They are connected to port E, pins 3 to 6. In order to read the value from pushbuttons the port E, pins 3 to 6 must be configured as inputs. When the pushbutton is pressed, the pin reads as logic high, otherwise the pin is left floating. However, we can use the pull-down resistors for port E, pins 3 to 6 to terminate pins to ground and un-float them. It is convenient again to prepare a function to initialize pins associated with pushbuttons, such function “SWITCHinit” is given below.

```
void SWITCHinit (void)  {
GPIO_InitTypeDef        GPIO_InitStructure;

  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE);

  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_6;
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_IN;
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_DOWN;
  GPIO_Init(GPIOE, &GPIO_InitStructure);
}
```

Please note that the ‘.GPIO_mode’ member is defined as ‘GPIO_Mode_IN’, and that ‘.GPIO_PuPd’ member is defined as ‘GPIO_PuPd_DOWN’. Members ‘GPIO_Type’ and ‘.GPIO_Speed’ have no meaning here since pins are used as inputs, and are not initialized.

A function for the configuration of pins associated with LEDs can be prepared similarly. LEDs are available at the STM32F4-Discovery board, and are wired to port D, pins 12 (green), 13 (orange), 14 (red), and 15 (blue). These pins must be configured as outputs, the function “LEDinit” is listed below.

```
void LEDinit (void)     {
GPIO_InitTypeDef        GPIO_InitStructure;

  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);                                 // 4

  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_OUT;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;                                      // 9
  GPIO_Init(GPIOD, &GPIO_InitStructure);
}
```

Please note the differences from the listing to initialize port E, pins 8 to 16 as outputs. Here the clock is enabled for port D, see line 4 of the listing. The direction of the signal flow is ‘OUT’, and both MOS transistors are used in push-pull configuration. The terminating resistors need not be specified, since this are output pins. The speed is reduced to 2MHz in line 9 thus saving power.

The rest of the program is straightforward, and is given below.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"
#include "dd.h"

void main (void)  {
char switches = 0;
  SWITCHinit();
  LEDinit();

  while (1) {
    switches = GPIOE->IDR;                                                    // 12
    if (switches & S370) LED_GR_ON; else LED_GR_OFF;
    if (switches & S371) LED_OR_ON; else LED_OR_OFF;
    if (switches & S372) LED_RD_ON; else LED_RD_OFF;
    if (switches & S373) LED_BL_ON; else LED_BL_OFF;
  };
}
```

The program starts with three familiar include statements. The fourth line includes one addition file with define statements for pushbuttons and LEDs. Next come the functions “SWITCHinit” and “LEDinit” as listed above but not shown here again to avoid redundancy.

Within the “main” function a variable ‘switches’ is declared and initialized first, next two functions for the configuration of pins for pushbuttons and LEDs are called. Finally the endless ‘while’ loop is entered, where the microprocessor reads the status of pushbuttons into variable ‘switches’, line 12. The name of the Input data register for port E is ‘GPIOE->IDR’. After reading the input the microprocessor checks the variable ‘switches’ against four different patterns, and lights the corresponding LED. It is worth to mention that commands written in capital letters are defined in file “dd.h”. Two special output registers are used; by setting a bit in register GPIOD->BSRRL the corresponding pin of port D is set, and by setting a bit in output register GPIOD->BSRRH the corresponding bit is reset. This can streamline setting or resetting of individual bits over writing a completely new pattern into the data output register, as well as avoids corrupting the state of other pins when only one pin is to be affected.

5. Alphanumeric LCD

The use of an alphanumeric LCD display with 16 characters, two lines is demonstrated.

5.1. Hardware – a commonly used 2X16 LCD

It is often necessary to present results of calculations to a user, and a liquid crystal display (LCD) may come handy. There are many different LCDs available; a simple LCD capable of presenting two lines with 16 characters in each line is used here. The display implements a standard bus and related datasheet and examples of driving such display can easily be found on the web. The LCD is connected to the microcontroller using a four-bit data bus and two-bit control bus. Additionally, power supply and contrast setting lines increase the number of wires at the connector to 10, all is available at connector K350, see Fig. 2.2.3.

Lower four pins of port D (0 to 3) are used to drive the four-bit data bus for the display, and pins 6 and 7 of port D are used to drive the control lines named Enable (E) and Register Select (RS). Since it is not planned to read from the LCD the control line Read/Write (RW), used to determine the direction of data transfer, is permanently connected to GND allowing only writing into the LCD.

The contrast of the characters displayed at the LCD depends of the position of trimmer potentiometer P350.

5.2. Software – the include file “LCD2x16.c”

The use of the LCD in a user program starts by its initialization, which must be performed at the beginning of the execution. Later the LCD can be used to display any alphanumerical symbols, strings of symbols or integer numbers of different lengths. A set of functions to utilize the display has been prepared, and is available in an include file named “LCD2x1616.c”. The functions include:

- LCD_init: This function is called at the beginning of the program to configure port D, lower four plus additional two pins, and the LCD itself using a four-bit interface. The function requires no parameters and returns no values.
- LCD_string: This function displays a string of characters (maximum 16 characters in length) at the LCD at the desired position, and accepts two parameters. The first is the string to be written (for instance “ABCD”), the second is the position at the LCD where the string should start. The first position in the upper line is indexed as 0x00, the last position in the first line is indexed as 0x0f. The first position in the second line is indexed as 0x40, the last position in the second line is indexed as 0x4f due to the hardware of the LCD used.
- LCD_uInt16: The function is used to display a 16-bit unsigned integer number at the LCD. The function requires three parameters. The first parameter is the unsigned integer to be displayed, the second parameter fixes the position on the LCD (as for the LCD_string function), and the third parameter is either 0 or 1. If 0 the number is shown including leading zeroes, if 1 the leading zeroes are removed. In both cases the number is right aligned.

- LCD_sInt16: The same as LCD_uInt16, but capable of displaying a 16-bit signed integer (15 bits and sign).
- LCD_uInt32: The same as LCD_uINT16 but capable of displaying a 32-bit unsigned integer number. Also requires three parameters (maybe not complete 32 bits, check the software).
- LCD_sInt32: The same as LCD_uInt32, but capable of displaying a 32-bit signed integer (31 bits & sign, maybe not complete 31 bits, check the software).
- LCD_sInt3DG: The same as LCD_uInt16, but capable of displaying only 3 digits.
- LCD_sInt4DG: The same as above, but capable of displaying 4 digits.

The file “LCD2x16.c” must be included at the beginning of the user program before the first call to any function from it. This is done by a “#include “LCD2x16.c” ” statement. In all integer displaying functions a conversion from binary to binary-coded-decimal in performed following the well-known algorithm from the web (search keyword: “shift and add 3 algorithm”).

The demo program to write a simple message (imitation of the measurement result) is given below.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"
#include "dd.h"
#include "LCD2x16.c"

void main (void){
int counter = 0;

  LCD_init();                                  // init LCD
  LCD_string("LCD test", 0x04);                // display title string

  while (1) {
    LCD_uInt16(counter++,0x40,0x01);    // write to LCD, unsigned, 16 bits
    //LCD_sInt16(counter++,0x40,0x01);    // write to LCD, signed, 16 bits
    //LCD_uInt32(counter++,0x40,0x01);    // write to LCD, unsigned, 32 bits
    //LCD_sInt32(counter++,0x40,0x01);    // write to LCD, signed, 32 bits
    //LCD_sInt3DG(counter++,0x40,0x01);   // write to LCD, unsigned, 3 digits
    for (int i = 0; i<1000000; i++) {}; // waste some time
  };
}
```

The program starts with the familiar set of four include statements, and the fifth include statements is added to allow the use of LCD functions.

The “main” function is executed, and a variable ‘counter’ is declared and initialized, then port D and the alphanumeric LCD module are configured and initialized by calling the function “LCD_init()”. Following the initialization a simple string is displayed using function “LCD_string()”, then the execution continues with the endless loop, where the content of the variable ‘counter’ is repeatedly displayed in the second line of the display. An empty for loop is added to slow down the execution of the loop.

A note: an LCD module can interpret messages sent from the microcontroller only if a well-defined protocol and timing is followed by the microcontroller. The protocol is fixed in functions enclosed in file “LCD2x16.c”. The timing associated with sending messages to the LCD module is implemented by executing empty for loops, where the number of iterations defines the delay. These delays were thoroughly tested for different LCD modules and optimized for speed, but might not be adequate for all LCDs. The resulting screen is either empty or garbled. If timing is too fast then it can be slowed down by increasing the numbers given in the define statements at the beginning of file “LCD2x16.c”, lines 5 to 7.

6. A stepper motor

The use of a stepper motor is demonstrated. A standard brushed or brushless motor can rotate; the bigger the applied voltage the faster it rotates, its use is simple and the torque can be high. However, it is very complex to move brushed or brushless motor for a given angle or to make it stay in a selected position; there is no torque to hold the rotor in a stationary position. Retaining the position can easily be achieved by a stepper motor, and they are primarily used where positioning is a main goal.

6.1. Hardware – a stepper motor basics

A stepper motor basically consists of four stationary electromagnets (A, B, C, D) and a permanent magnet (R) which serves as a rotor, and has two magnetic poles N and S. Consider a current flowing through only one of the electromagnets (A, B, C or D) at a time. Figure 6.1 gives corresponding orientations of the rotor made of permanent magnet in the middle. It is to be emphasized that the rotor remains in a given orientation as long as the current is flowing through the same electromagnet, and that the user knows the orientation of the rotor since the user is the one to turn-on a selected electromagnet.

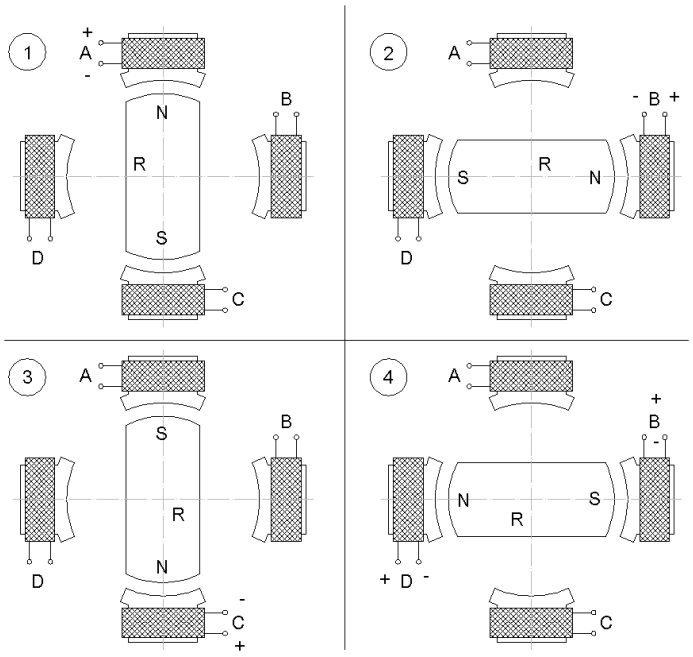


Figure 6.1: Four (1, 2, 3, and 4) possible orientations of the rotor versus the active electromagnet (A, B, C, D)

There are two kinds of stepper motors as far as the wiring of the electromagnets is concerned.

- The first, weaker version, has one side of each electromagnet connected together and exposed to the user. The other side of each electromagnet is also exposed to the user. Such stepper motor has **five wires** (sometimes **six**, since the common side of all electromagnets is available

through two separate wires). The common wire is supposed to be tied to a power supply, and other four wires are to be connected to the ground using four switches as in Fig. 6.2. If switches are controlled by four digital signals in a way that an individual bit of a four-bit binary number controls individual switch (1 turns it ON, 0 turns it OFF), then a sequence of numbers 0001b, 0010b, 0100b, 1000b, 0001b … will rotate the rotor in one direction, while the reverse sequence will rotate the rotor in the other direction. The transition from one number to the other number must be delayed enough to allow the movement of the rotor.

The current flows through one electromagnet at a time therefore 3 out of 4 of the electromagnets are wasting space inside the motor. In order to boost the use of space and with this the torque of the motor, two adjoining electromagnets can be powered simultaneously. This roughly doubles the torque (and current consumption), and is implemented by OR-ing the adjoining numbers from the sequence above to get: 0011, 0110, 1100, 1001, 0011 … for rotation in the same direction.

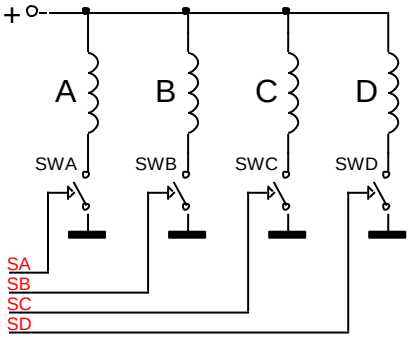


Figure 6.2: The connection of the five-wire stepper motor

- There is also the second, stronger version of a motor which has two transversely positioned electromagnets connected in series, giving altogether **four wires** to drive two pairs of electromagnets. The connection of such motor is shown in Fig. 6.3. It requires more elaborate combination of switches called “a bridge connection” and the current can flow in one of two possible directions through each of the two pairs of electromagnets. The sequence of signals required to drive the switches in any version of a motor is the same and we will not elaborate here anymore on this. The only thing to emphasize is that both electromagnets are conducting current permanently in this motor; only the direction of current changes to move the rotor. This results in the best utilization of the space inside the motor and maximum torque.

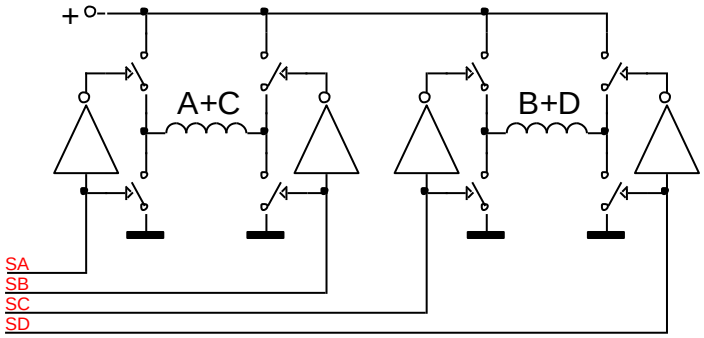


Figure 6.3: The connection of a four-wire stepper motor

A chip capable of driving a stepper motor is included on the BaseBoard. It is the L293D, an industrial standard driver for small power motors which includes four individually controllable halves of a bridge connection. Maximum current for one half of a bridge is almost 1A and the chip can work with a power supply from 5V to 36V. The outputs from the chip are initially disabled, but can be enabled using two enable lines. All inputs lines to the chip can be driven directly from the microcontroller and protective diodes against voltage spikes caused by a motor are integrated on the chip. The stepper motor requires more current than available from the STM32F4_Discovery board, so an external power supply is

mandatory. The external power supply can be connected to the BaseBoard only to supply the L293D, the voltage should be set to conform to the requirements of the stepper motor, but remain in the safe range for the L293D.

From the discussion above a stepper motor will make one turn in four consecutive steps. However, such division is much too coarse for positioning. A typical stepper motor has to make 200 steps to make one complete circle; therefore one step rotates the rotor for 1.8 degrees. This is done by additional shaping of the core of the electromagnet and the rotor. Consider the drawing in Fig. 6.4. This time the rotor is not a permanent magnet, but a solid iron. When electromagnet A is energized, the grooves on the rotor align with the grooves on the core of the electromagnet A. When electromagnet A is turned off and electromagnet B is turned on, the rotor changes its position to align the groves with electromagnet B; the rotor rotates for ¼ of a groove. Next electromagnet C is turned on and electromagnet B off. The result is an additional rotation for ¼ of a groove. The same rotation takes place for the last electromagnet D, therefore in four consecutive steps the rotor turns for one groove. The number of steps for one full turn is defined by 4 times the number of grooves.

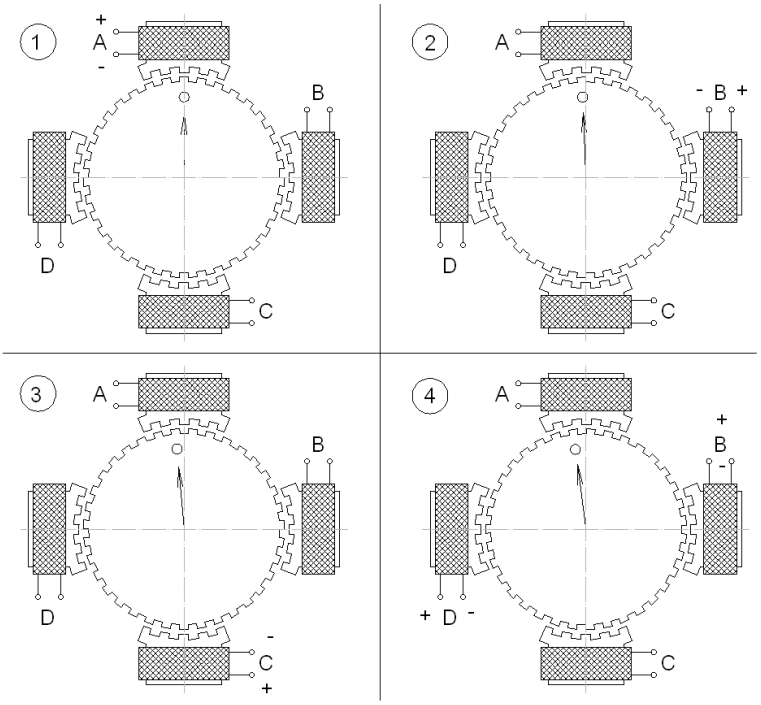


Figure 6.4: Stepper motor with more steps per revolution

6.2. Software to drive the stepper motor

In order to use a stepper motor it must first be connected following either of the figures 6.2 or 6.3. Since port C controls the input to the driver L293D, corresponding pins PC6 to PC9 and PC11 must be configured as push-pull outputs. Here pin PC11 serves as an enable signal for the driver L293D, and must be set to high. Other pins serve as control lines for the driver chip, their values follow the pattern stated above.

The function for the initialization of the port C to handle the stepper motor driver is given below.

```
void STEPPERinit (void)   {
GPIO_InitTypeDef        GPIO_InitStructure;

  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);                                    // 4
```

```
  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_6 | GPIO_Pin_7;                    // 5
  GPIO_InitStructure.GPIO_Pin  |= GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_11;     // 6
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_OUT;                             // 7
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;                             // 8
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;                           // 9
  GPIO_Init(GPIOC, &GPIO_InitStructure);                                     // 10
}
```

The function follows the standard form presented in chapter 4. Line four enables the clock for port C, and line 10 calls the function to configure the port C, selected pins, with the desired options. The options are written into the initializing structure GPIO_InitStructure in lines 5 to 9.

A demonstration program is prepared. The program configures necessary pins of the microprocessor, and then enters the infinite while loop where it periodically reads the state of pushbuttons. If pushbutton S370 is pressed then the motor is rotated clockwise. If pushbutton S371 is pressed then the motor is rotated counterclockwise. The loop is terminated by an empty loop to slow-down the execution. The current position of the motor is written to the LCD screen.

The pattern to drive the motor is initialized in an array at the global declare section of the program. The pattern in this array might need to be adjusted to the order of wires connected to the driver L293D.

The complete listing of the program is given below. The function for the initialization of the port C “STEPPERinit()” was given above, and the function for the initialization of pins needed for pushbuttons “SWITCHinit()” was given in chapter 4.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"
#include "LCD2x16.c"
#include "dd.h"

int steps[4] = {BIT_6, BIT_7, BIT_8, BIT_9};
//int steps[4] = {BIT_6 | BIT_7, BIT_7 | BIT_8, BIT_8 | BIT_9, BIT_9 | BIT_6};
int ptr_steps = 0;

void main (void)  {              //
char switches = 0;

  STEPPERinit();
  SWITCHinit();

  LCD_init();                           // init LCD
  LCD_string("Stepper motor", 0x01);    // display title string
  GPIOC->ODR |= GPIO_Pin_11;            // enable motor driver

  while (1) {                   //
    switches = GPIOE->IDR;
    if (switches & S370)  ptr_steps++;
    if (switches & S371)  ptr_steps--;
    GPIOC->ODR &= ~0x3c0;
    GPIOC->ODR |= steps[ptr_steps & 0x03];

    LCD_sInt16(ptr_steps,0x40,0x05);    // write to LCD, signed, 16 bits
    for (int i = 0; i<1000000; i++) {}; // waste some time

  };
}
```

Note that due to the pushbutton pressed a variable ‘ptr_steps’ is modified. The last two bits of this variable are used as a pointer to the array with predefined patterns to rotate the motor in the correct position. This trick simplifies the rotation and simultaneous following of the position. It also allows negative positions to be interpreted correctly.

7. Digital to analog converter

The use of a DAC is demonstrated.

7.1. Hardware – a DAC block

There are two 12-bit digital to analog converters (DAC) available. A principal block diagram for one of the two identical DACs is given in Fig. 7.1 (Fig. 64, RM0090, pg. 431). A letter 'x' can be '1' or '2', depending on the DAC in question.

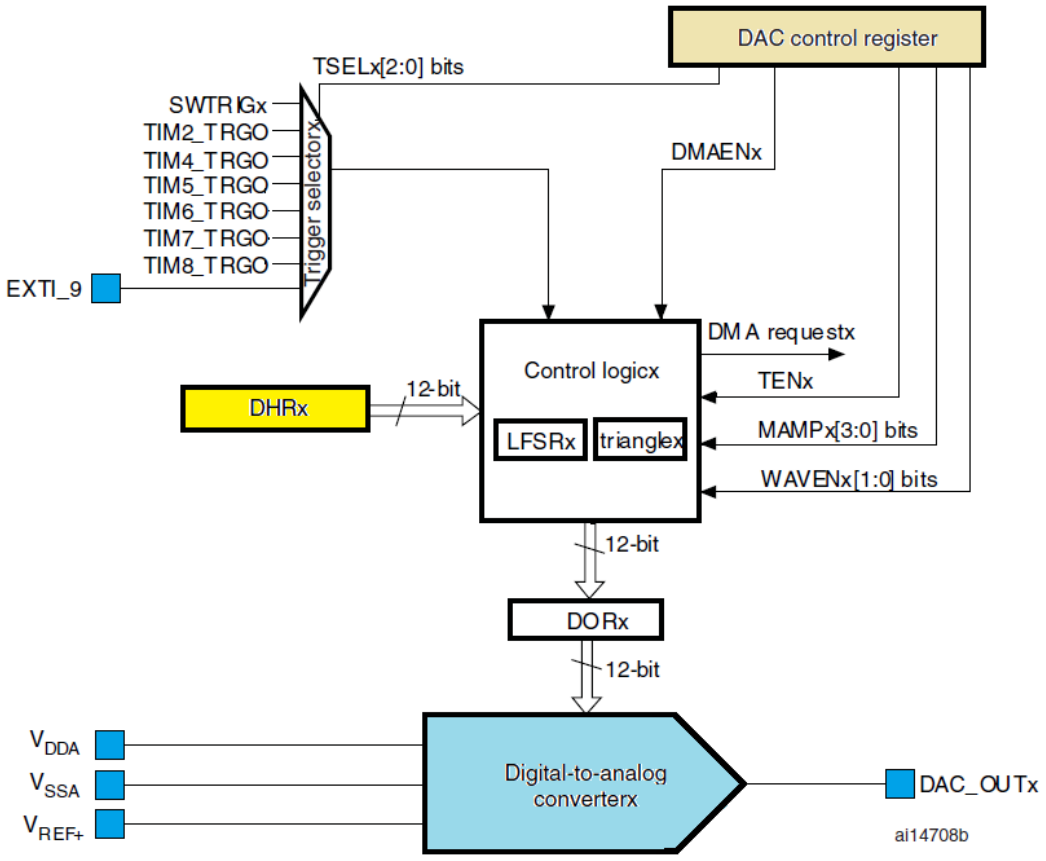


Figure 7.1: The block diagram of a single DAC

The actual DAC is shown at the bottom. Its operation is supported by power supply lines V_{DDA} and V_{SSA}, and the reference signal V_{REF+}, all coming from pins of the microcontroller on the left. The output of the DACx is available at a pin to the right DAC_OUTx. The user program loads data (to be converted to analog signal) into the data holding register DHRx from where it is copied into register DORx under a hardware control to be instantly converted into analog signal by the DAC. The hardware enclosed in block Control logic defines the moment of copying.

The register DHRx is a 32-bit wide register, but can be accessed as a whole (DAC_DHR12RD, 12 bits for each DAC, data right aligned in both halves of the 32-bit register) or each half separately (DAC_DHR12R1 and DAC_DHR12R2, 12 bits for each DAC1 or DAC2, data right aligned in a 32-bit register). Each half of the register DHR is responsible for one DAC, see the details in RM0090, page 445 to 451.

The moment of transferring the data from register DHR into DORx can be automatic, but can also be defined by software (SWTRIGx), hardware (TIMx_TRGO) or external signals (EXTI_9), as selected by the multiplexor on the left above the register DHRx. Such organization allows the synchronization of signals out of the two DAC with a selected event. There is also a possibility to transfer the data using a direct memory access.

Additional hardware is provided in Control logicx box to generate noise or triangle wave signal to be added to the content of the register DHR. This will not be used in this experiment.

7.2. Software to test the DAC

Several bits distributed along control registers for the DAC are responsible for the behavior of the two DACs, but the programmer should know their location and detailed function. To ease the programming the CMSIS library will be used. The functions to actually configure the operation of DACs are stored in the source file “stm32f4x_dac.c”, and the definitions and data structures to go along are stored in the header file “stm32f4x_dac.h”.

The recipe to use the DAC is given in the source file under section ‘How to use this driver’:

- Configure pins where the two DAC outputs are connected, these are port A, pins 4 and 5 for STM32F407VG microcontroller. First enable the clock for port A, then put pins 4 and 5 in analog mode, and disable terminating resistors.
- Enable the clock for the DAC block.
- Configure both DACs.
- Enable both DACs.

The DACs can be used afterwards by simply writing new digital value into the DHR register; corresponding analog value will appear at port A, pins 4 and 5.

The recipe is implemented in a function “DACinit()” listed below.

```
void DACinit (void)     {
GPIO_InitTypeDef        GPIO_InitStructure;                                          // 2
DAC_InitTypeDef         DAC_InitStructure;                                           // 3

  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);                              // 5
  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_4 | GPIO_Pin_5;                           // 6
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AN;                                      // 7
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;                                  // 8
  GPIO_Init(GPIOA, &GPIO_InitStructure);                                             // 9

  RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);                                // 11
  DAC_InitStructure.DAC_Trigger        = DAC_Trigger_None;                           // 12
  DAC_InitStructure.DAC_WaveGeneration = DAC_WaveGeneration_None;                    // 13
  DAC_InitStructure.DAC_OutputBuffer   = DAC_OutputBuffer_Enable;                    // 14
  DAC_Init(DAC_Channel_1, &DAC_InitStructure);                                       // 15
  DAC_Init(DAC_Channel_2, &DAC_InitStructure);                                       // 16

  DAC_Cmd(DAC_Channel_1, ENABLE);                                                    // 18
  DAC_Cmd(DAC_Channel_2, ENABLE);                                                    // 19
}
```

Two data structures are needed, and these are declared in lines 2 and 3. The first data structure is required to initialize port A with a function call in line 9. The second data structure is required to initialize DAC blocks with a function calls in lines 15 and 16.

Line 5 enables the clock for port A, and lines 6 to 8 fill the data structure that will serve to initialize the port. Note that the setting is applied to pins 4 and 5 only, and that the selected mode is analog ("GPIO_Mode_AN"). Terminating resistors at these pins are disabled ("GPIO_PuPd_NOPULL").

Line 11 enables the clock for the DAC block. Next comes the initialization of members of the second data structure. There are four members altogether, as can be seen in a header file "stm32f4x_dac.h", lines 54 to 69:

- Member '.DAC_Trigger' tells the hardware enclosed in block Control logic when to copy data from the Data holding register to Data output register. In our case the hardware will copy data immediately after to software sends new value into register DHR, therefore option 'DAC_Trigger_None' is used. All available options are listed in the header file, lines 81 to 91. Basically, signals connected to the multiplexor, Fig. 7.1 top left, can be used to initiate the copying. These signals are coming either from timers inside the microcontroller, from external sources, or can be defined by software. In any case option other than the one used in the listing above assures an update of DAC outputs which is synchronous with the selected event.
- Member '.DAC_WaveGeneration' allows the hardware enclosed in block Control logic to autonomously generate a triangular signal or noise and add it to the value specified in register DHR. Such option is convenient for special measurements, but will not be used in this example. To disable the option this member is initialized to 'DAC_WaveGeneration_None', other options are listed in the header file, lines 111 to 113.
- Member '.DAC_LFSRUnmask_TriangleAmplitude' can be used to define properties of the added triangular or noise signal. Since in our case these were not used, we do not need to initialize this member.
- Member '.DAC_OutputBuffer' determines the use of an additional buffer stage between the DAC and the pin of microprocessor. When this buffer is used the output impedance is reduced allowing stronger loading of the DAC, but increasing the power consumption of the microcontroller. In this example the output buffers are enabled by initializing the member with 'DAC_OutputBuffer_Enable'.

This data structure is then used to configure both DACs in two consecutive function calls "DAC_Init()", lines 15 and 16. Note that the first argument of the function specifies the DAC to be affected, and the second argument is a pointer to the data structure.

The next two lines are used to enable both DACs using the command "DAC_Cmd()" with two arguments again. The first specifies the DAC in question, and the second commands the enabling.

The demo program for the two DAC generates two sawtooth signals by writing consecutively increasing numbers to Data holding register for DAC1 ('DAC->DHR12R1') and consecutively decreasing numbers to Data holding register for DAC2 ('DAC_DHR12R2'). These two writes could be replaced by two CMSIS function calls "DAC_SetChannel1Data()", but the version used in this example is faster (and less portable). The listing of the program is given below. Note the additional include file "stm32f4x_dac.c".

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_dac.c"
#include "stm32f4xx_gpio.c"

int main ()  {
```

```
unsigned int j;

  DACinit();

  while (1) {
    DAC->DHR12R1 = j;           // up ramp
    DAC->DHR12R2 = 0xfff - j;   // down ramp
    j = (j + 1) & 0x0fff;
  };
};
```

8. Analog to digital converter

The use of an ADC is demonstrated.

8.1. Hardware – a ADC block

There are three 12-bit analog to digital converter (ADC) blocks available in STM32F407VG microcontroller. The conversion time for each ADC block is about 1µs. There is a rich variety of possible ADC configurations, and here we demonstrate the simplest one to measure two voltages simultaneously using two ADCs.

A simplified block diagram for one ADC block is given in Fig. 8.1 (excerpt from Fig. 44, RM0090, page 387). The complete ADC block includes two more ADCs with similar features and additional settings.

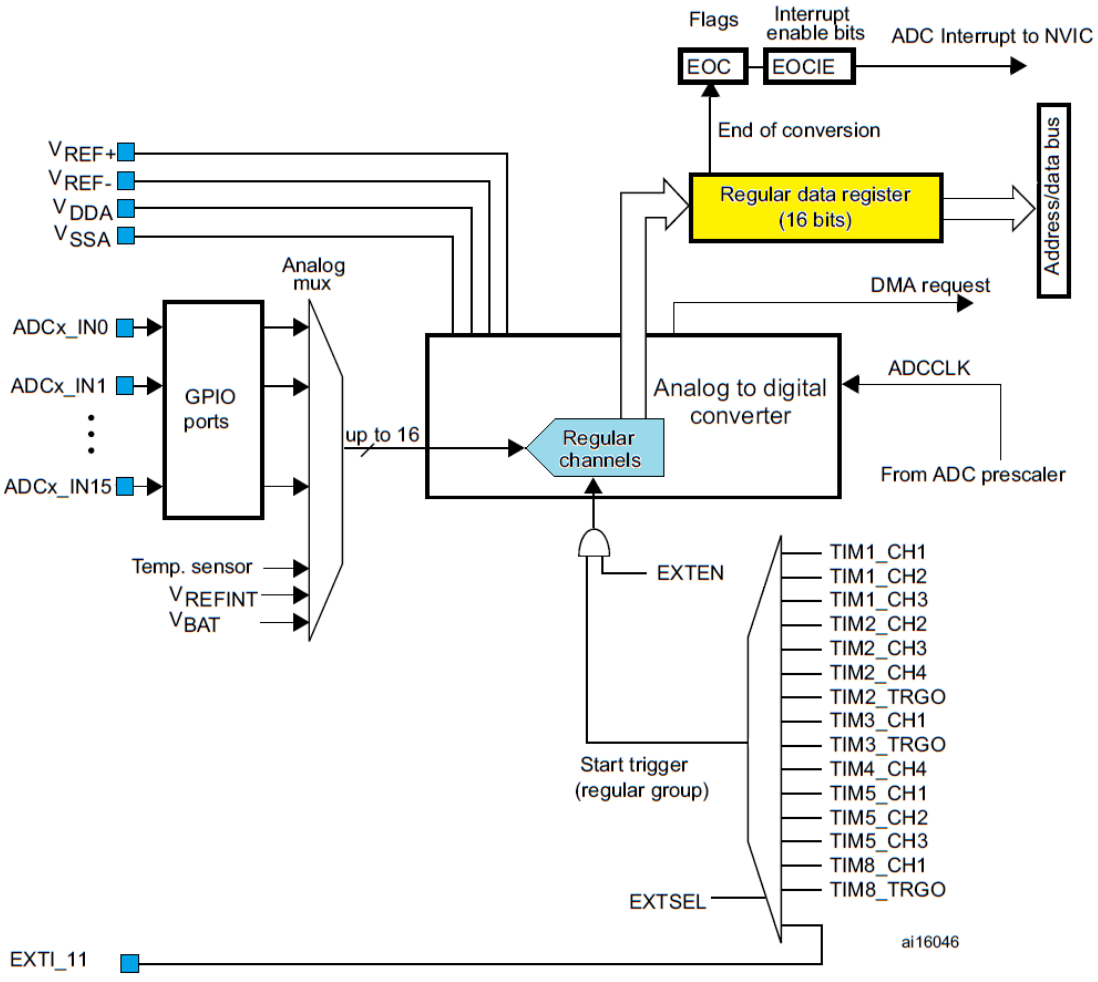


Figure 8.1: The block diagram of a single ADC

The actual ADC is located in the center of the figure. Its operation is supported by power supply lines V_{DDA} and V_{SSA}, and the reference lines V_{REF+} and V_{REF-}, all coming from pins of the microcontroller. Additionally, a clock signal ADCCLK is required, and is generated in the microcontroller. The analog input signals are connected to the designated microcontroller pins ADCx_INxx and are fed to the ADC through a multiplexer named Analog mux. There are altogether 16 pins available to connect analog signals to, and the multiplexer houses additional three inputs to allow the measurement of the chip temperature, power and reference lines. Appropriate input pin is selected by setting bits (five of them) in one of the ADC registers (ADCx_SQR3, if one wants to do it manually). It should be noted that analog signals do not mix well with digital signals, and that the pin used for input must be configured as analog.

The ADC writes the result of conversion into a Regular data register (ADCx_DR). Here x stands for the index of the ADC used (1, 2 or 3).

The ADC block includes hardware to support up to 16 consecutive conversions at pins declared in advance without the assistance from the microcontroller; when one intends to implement consecutive conversions the number of consecutive conversions and the input pins used must be stated before initiating the conversion. However, there is only one register to hold a result of conversion, so a user program must take care of moving consecutive results from the Regular data register before next conversion is finished or the previous result will be lost. The DMA (direct memory access) hardware can be utilized to move results in time.

8.2. Software to test the ADC

The ADC block can be configured by manipulating bits in designated registers of the ADC block. However, this requires a detailed knowledge on the hardware of the ADC, and it is much simpler to use functions available in CMSIS library. The functions that actually manipulate bits are stored in the source file “stm32f4_adc.c”, and definitions and data structures to accompany these functions are stored in the header file “stm32f4x_adc.h”. The first of them must be included from the user program.

A brief recipe for the use of two or more ADCs is again given at the beginning of the source file:

- Configure pins that will be used to input analog signals. They must be designated as analog pins, there must be no terminating resistors.
- Enable clock for ADCs used.
- Configure the control circuitry valid for all three ADCs within the STM32F407VG, use function “ADC_CommonInit()” and the related data structure.
- Configure the use of regular channels for every ADC individually using function “ADC_Init()” and the related data structure. Connect ADCs with input signals.
- Enable ADC(s) using function “ADC_Cmd()”.

The result of conversion can be obtained by reading the content of register ADCx->DR, where x stands for either 1, 2 or 3. Alternativelly, and to enhance portability of the program, a CMSIS function “ADC_GetConversionValue()” can be used.

The recipe for configuration of two ADCs to sample two input signals simultaneously is implemented in the function and listed below.

```
Void ADCinit_SoftTrigger(void)  {
              // 1
GPIO_InitTypeDef        GPIO_InitStructure;                                          // 2
ADC_InitTypeDef         ADC_InitStructure;                                           // 3
ADC_CommonInitTypeDef   ADC_CommonInitStructure;                                     // 4

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1 | RCC_APB2Periph_ADC2,  ENABLE);         // 6
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);                              // 7
```

```
  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_1 | GPIO_Pin_2;                                  // 9
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AN;                                            // 10
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;                                        // 11
  GPIO_Init(GPIOA, &GPIO_InitStructure);                                                   // 12

  ADC_CommonInitStructure.ADC_Mode             = ADC_DualMode_RegSimult;                   // 14
  ADC_CommonInitStructure.ADC_Prescaler        = ADC_Prescaler_Div2;                       // 15
  ADC_CommonInitStructure.ADC_DMAAccessMode    = ADC_DMAAccessMode_Disabled;               // 16
  ADC_CommonInit(&ADC_CommonInitStructure);                                                // 17

  ADC_InitStructure.ADC_Resolution             = ADC_Resolution_12b;                       // 19
  ADC_InitStructure.ADC_ScanConvMode           = DISABLE;                                  // 20
  ADC_InitStructure.ADC_ContinuousConvMode     = DISABLE;                                  // 21
  ADC_InitStructure.ADC_ExternalTrigConvEdge   = ADC_ExternalTrigConvEdge_None;            // 22
  ADC_InitStructure.ADC_ExternalTrigConv       = ADC_ExternalTrigConv_T1_CC1;              // 23
  ADC_InitStructure.ADC_DataAlign              = ADC_DataAlign_Right;                      // 24
  ADC_InitStructure.ADC_NbrOfConversion        = 1;                                        // 25
  ADC_Init(ADC1, &ADC_InitStructure);                                                      // 26
  ADC_RegularChannelConfig(ADC1, ADC_Channel_1, 1, ADC_SampleTime_3Cycles);                // 27

  ADC_Init(ADC2, &ADC_InitStructure);                                                      // 29
  ADC_RegularChannelConfig(ADC2, ADC_Channel_2, 1, ADC_SampleTime_3Cycles);                // 30

  ADC_Cmd(ADC1, ENABLE);                                                                   // 32
  ADC_Cmd(ADC2, ENABLE);                                                                   // 33
}
```

The function starts with the declaration of data structures, three are required for three different function calls. Lines 6 and 7 hold function calls to enable the clock signals for ADC1, ADC2 and port A.

Analog voltages to be measured are connected to pins 1 and 2 of port A, see schematic diagram of the BaseBoard for available pins. These two pins must be configured as analog without terminating resistors, so lines 9 to 11 initialize members of the data structure 'GPIO_InitStructure' to correct values, and line 12 calls the function to actually configure the port A.

The hardware that is common to all three ADCs is configured next, this is done in line 17 with a call to a function "ADC_CommonInit()", the function itself is located in the source file "stm32f4xx_adc.c". This function requires a pointer to the data structure 'ADC_CommonInitStructure' as the only argument, and the structure itself is initialized in lines 14 to 16. There are four members of this structure.

- The first member '.ADC_Mode' defines how the operation of the three ADC is synchronized. Their operation can be independent ('ADC_Mode_Independant'), and each ADC is started independently of others. Their operation can be simultaneous ('ADC_TripleMode_RegSimult'), and all three ADCs are started using a single trigger event. Their operation can be interleaved ('.ADC_TripleMode_Interl'), and each trigger event starts the first ADC, the other two follow after a fixed time interval, one after another. There are similar options for the use of two ADCs, we have selected the option '.ADC_DualMode_RegSimult': two ADCs are used, and they are started simultaneously on the trigger event. Other options can be found in the header file "stm32f4xx_adc.h", lines 115 to 127, and are described in reference manual RM0090.
- The second member '.ADC_Prescaler' selects the clock frequency ADCCLK. This clock determines the conversion time, and is not the same as the clock enabled in line 6 of the function, see the reference manual RM0090, pg. 388, for the difference. The clock ADCCLK is common for all ADCs, and can be either ½, ¼, 1/8 or 1/16 of the microprocessor internal clock f_{PCLK} frequency, which equals to 42 MHz in our case. Since the ADC can use clock frequency up

to 30 MHz it is safe to use the least division factor of ½. The correct option is then 'ADC_Prescaler_Div2', others can be found in the header file, lines 149 to 152.

- The third member '.ADC_DMAAccessMode' defines the use of DMA hardware to move results of conversion from the Regular data register into a reserved part of the microprocessor RAM. We are not going to use this option, so the correct keyword is 'ADC_DMAAccessMode_Disabled'. Other options are listed in header file, lines 165 to 168, and the explanation is given in the reference manual RM0090, chapter 13.
- The fourth member defines the time interval between consecutive starts of ADCs if the interleaved mode is selected. We do not use this mode, so this member need not be defined.

Settings for each ADC individually are configured next. We are going to use two ADCs, and both should work the same way; we therefore call the function "ADC_Init" to initialize the ADC two times in lines 26 and 29, once for 'ADC1' and once for 'ADC2'. The function requires a pointer to the data structure 'ADC_InitStructure', and parameters for the two calls are the same. The data structure 'ADC_InitStructure' has seven members, they are initialized in lines 19 to 25.

- An ADC can operate with different resolutions; lower resolutions allow faster conversion. For this example the best resolution of 12 bits is required, and the corresponding conversion time is less than 1μs. This is selected by initializing the member 'ADC_Resolution' with 'ADC_Resolution_12b'. Other options can be found in the header file, lines 223 to 226.
- An ADC starts a measurement on a trigger event. Within the measurement it can sample one single signal connected to one pin, or it can scan a predefined set of signals connected to several pins. We need to measure a single signal for each trigger event, and this is achieved by disabling the scanning option in line 20.
- An ADC can start a measurement on trigger event and then wait for next trigger event, or can continuously repeat the measurement without repeating the trigger event. We want a single conversion, and this is achieved by disabling the continuous conversion mode in line 21.
- The fourth member '.ADC_ExternalTrigConvEdge' configures the edge of the signal to trigger the conversion. The edge can be rising, falling, both of them, or none. We want to start a conversion using a software command here, so option 'ADC_ExternalTrigConvEdge_None' is used in line 22.
- The fifth member '.ExternalTrigConv' selects one of the 16 signals to be used as trigger event by configuring the multiplexor bellow the ADC, Fig. 8.1. All possible options are listed in the header file, lines 256 to 271. It is not necessary to initialize this member in this example since we disabled the external trigger in line 22 and by this selected the software trigger, but the line 23 remains there for reference.
- The sixth member '.ADC_DataAlign' determines the alignment of the result in the Regular data register; it can be either left or right. The right alignment is selected by 'ADC_DataAlign_Right'.
- The seventh member 'ADC_NbrOfConversion' determines the number of consecutive conversions to be performed by one ADC within one measurement. Since we do not want to scan several analog signals, this member is initialized to 1.

Using this data structure two ADCs are initialized in lines 26 and 29.

The ADCs need to be connected through the multiplexor with sources of analog signals. These are pins 1 and 2 of port A, designated as ADC_Channel_1 and ADC_Channel _2, see the pinout table in the datasheet DM00037051, page 47. This connection can be achieved by setting of several bits residing in ADC control registers, which is best achieved by a call to a CMSIS function "ADC_RegularChannelConfig()", its brief description can be found in the source file, line 636 and on. The function requires 4 arguments.

- The first argument is the name of the ADC to be configured, can be ADC1, ADC2 or ADC3.

- The second argument is used to put the multiplexor in front of the ADC in correct state, and simply declares the number of channel to be measured. It can be any value from 'ADC_Channel_0' to 'ADC_Channel_18'.
- The third argument specifies the order of conversion if a sequence of conversions is initiated. This example does not use the sequenced conversions, and this argument is set to 1 since this is to be the first (and only) conversion.
- The fourth parameter defines the sampling time of the ADC, and can be minimum 3 clock cycles ('ADC_SampleTime_3Cycles') and maximum 480 clock cycles ('ADC_SampleTime_480Cycles'). All possible values are listed in the header file, lines 359 to 366.

Using these arguments the two lines 27 and 30 for individual configuration of ADCs are written.

The last thing is to enable ADCs; this is achieved by two calls in lines 32 and 33, the applicable CMSIS functions are "ADC_Cmd(ADCx, ENABLE)", where x stands for either 1 or 2.

The demo program has been prepared, and its listing is given bellow. The program uses ADC1 and ADC2 to measure voltages at pins 1 and 2, port A. The listing starts with the inclusion of necessary files, note the "stm32f4xx_adc.c", and the executable part starts with the initialization of both ADCs by calling the function "ADCinit()". This is followed by the initialization of the LCD screen, and writing of the introductory strings to the LCD.

The endless 'while' loop is executed next. Within the loop the conversion on ADC1 is initiated by calling the CMSIS function "ADC_SoftwareStartConv(ADC1)", and since the two ADCs were configured as simultaneous, this also starts the conversion on ADC2. Then some time is wasted allowing ADCs to finish the conversion, and both results of conversions are read from two Regular data registers of both ADCs and displayed. Please note that the reading of results is done by a direct access to the hardware; the name of the regular data register is stated as 'ADC1->DR' and 'ADC2->DR', which does not promote porting or CMSIS. There is a CMSIS function to retrieve the result available, its name is "ADC_GetConversionValue()", but it is not used here for simplicity and the speed of operation.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_adc.c"
#include "stm32f4xx_gpio.c"
#include "LCD2x16.c"

int main ()  {
unsigned int j;

  ADCinit_SoftTrigger();

  LCD_init();
  LCD_string("ADC1=", 0);
  LCD_string("ADC2=", 0x40);

  while (1) {
    ADC_SoftwareStartConv(ADC1);
    for (j = 0; j<10000000; j++){};      // waste some time
    LCD_uInt16((int)ADC1->DR, 0x06, 1);  // write result on LCD, 1st line
    LCD_uInt16((int)ADC2->DR, 0x46, 1);  // write result on LCD, 2nd line
  };
}
```

9. Timer – counting pulses

The microcontroller houses several timers that can be used for different purposes. This example shows the use of timer for counting of pulses that are applied to the pin of the microcontroller.

9.1. The hardware – a Timer

There are 14 timers included in the STM32F407VG. Their key properties are listed in Fig. 9.1 (DM00037051, pg. 30, 31). Detailed properties and the description of each group of timers can be found in RM0090. We will use Timer2 in this experiment to count pulses applied to a pin on the microcontroller and show current content of the counter on the LCD. There will be three pushbuttons used to initialize the content of the counter, to start counting, and to stop counting.

Timer type	Timer	Counter resolution	Counter type	Prescaler factor	DMA request generation	Capture/ compare channels	Complementary output	Max interface clock (MHz)	Max timer clock (MHz)
Advanced-control	TIM1, TIM8	16-bit	Up, Down, Up/down	Any integer between 1 and 65536	Yes	4	Yes	84	168
General purpose	TIM2, TIM5	32-bit	Up, Down, Up/down	Any integer between 1 and 65536	Yes	4	No	42	84
	TIM3, TIM4	16-bit	Up, Down, Up/down	Any integer between 1 and 65536	Yes	4	No	42	84
	TIM9	16-bit	Up	Any integer between 1 and 65536	No	2	No	84	168
	TIM10, TIM11	16-bit	Up	Any integer between 1 and 65536	No	1	No	84	168
	TIM12	16-bit	Up	Any integer between 1 and 65536	No	2	No	42	84
	TIM13, TIM14	16-bit	Up	Any integer between 1 and 65536	No	1	No	42	84
Basic	TIM6, TIM7	16-bit	Up	Any integer between 1 and 65536	Yes	0	No	42	84

Figure 9.1: The timers included in STM32F407 and their key properties

Timers TIM1 and TIM8 use 16-bit counters and are the most complex timers of all timers included in the microcontroller. Timers TIM2 and TIM5 are 32-bit versions of TIM1/TIM8, but have less

hardware included and therefore less options. Timers TIM3 and TIM4 are 16-bit versions of TIM2/TIM5. Timers TIM9, TIM10 and on are stripped-down versions of timer TIM4, and so on.

A simplified block diagram for timer TIM2 is given in Fig. 9.2 (reference manual RM0090, Figure 134, page 577). The content of the counter CNT (yellow) is available in register TIM2_CNT.

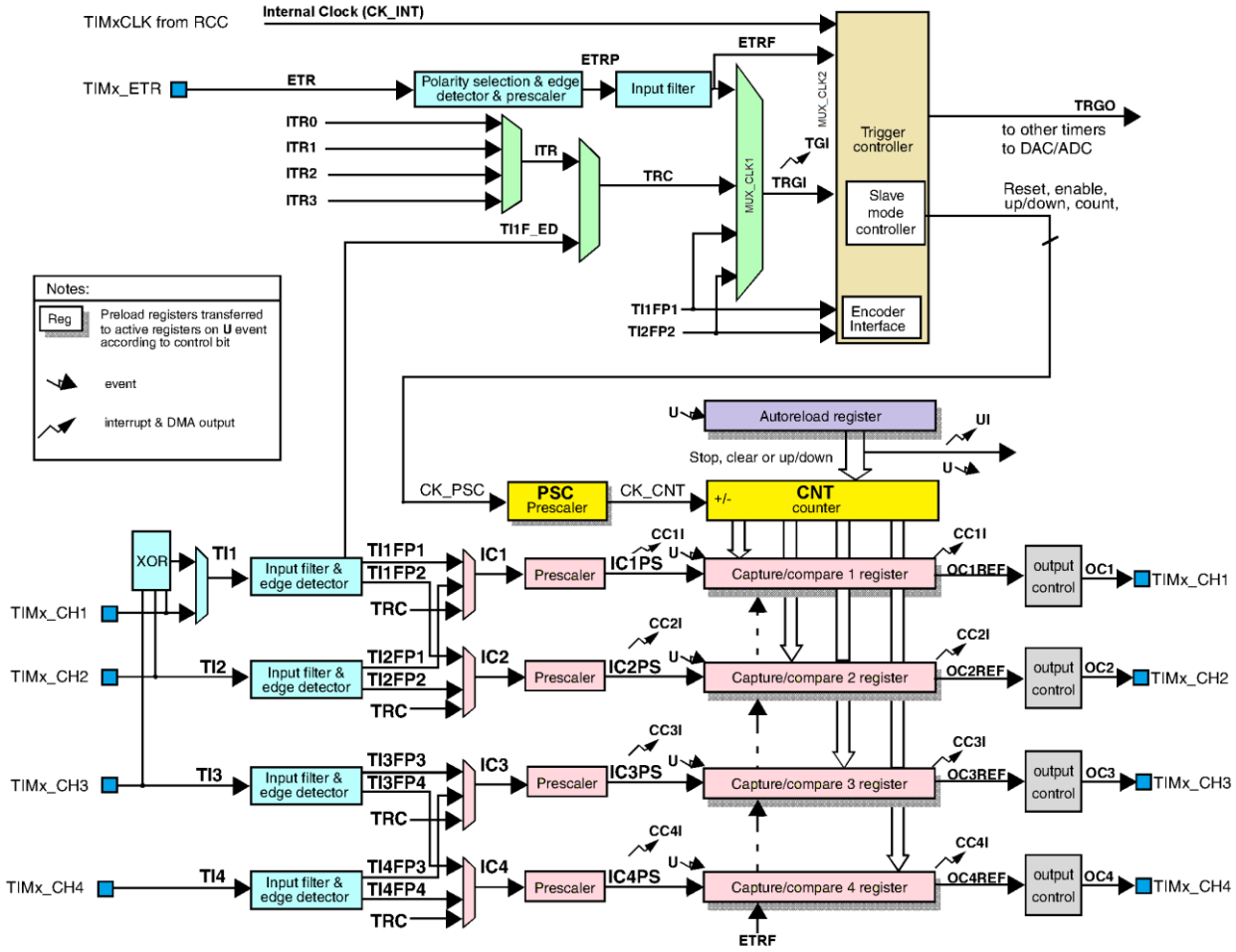


Figure 9.2: The block diagram for timers TIM2 and TIM5. Counter: yellow boxes, clock select: light green boxes, input signals conditioning: light blue boxes, output control: grey boxes.

The counter advances on transitions of the clock signal CK_PSC. This clock signal comes through a set of modules (light blue boxes and light green multiplexors) at the top of Fig. 9.2. Several signals are offered to become the source of the clock signal, and multiplexors select one of them as the 'CK_PSC'. The user can configure multiplexors to pass signals connected to pins marked TIMx_CH1 to TIMx_CH4 or TIMx_ETR. Alternatively, multiplexors can be configured to pass internally generated signals ITR0 to ITR3 or even internal clock signal CK_INT as a clock CK_PSC. The counter can also be fed by some internally derived signals like TI1FP1, TI2FP2, and then some. Details are given in reference manual RM0090, chapter 18.

Due to the availability of pins at the STM32F4-Discovery board and the limitations imposed by the BaseBoard we will use pin 15, port A, and designate it as a source of signal TIM2_ETR, which will in turn be routed through multiplexors to become the clock signal CK_PSC. This pin is available at the connector K406 at the edge of the BaseBoard.

The timer block houses additional sub-blocks, which will not be used in this experiment on counting. These blocks include prescaler modules, capture and compare modules, output modules and reload logic; this will be used in subsequent experiments.

9.2. The software to test the counting with Timer 2

Several bits distributed along registers within the block of Timer 2 are responsible for the configuration of the block and with this for its operation. Again, CMSIS functions stored in the source file “stm32f4xx_tim.c” are used to ease the correct configuration, and data structures and definitions are taken from the header file “stm32f4xx_tim.h”.

Two blocks are to be configured, these are the port A to pass the signal from its pin 15 to the Timer 2, and the Timer 2 to accept this signal, route it to the input of the counter and count it.

Let us deal with the configuration of the port first. A pin of a port is normally used to pass digital signals into or out of the microcontroller. It can be configured for analog signals, as we have seen in chapter on DAC and ADC. But a pin can also be used to pass special signals to or from peripherals which are built into the microcontroller; in this case we say the pin is assigned to an alternate function. Not all pins can pass signals from all peripherals, every pin can be assigned one of 16 alternate functions. The complete list of alternate functions available for every pin is given in table 9, pg. 60 and on, in the datasheet of the microcontroller (DM00037051). From this table one can see that port A, pin 15, can be used as a regular IO for digital signals, but it can also be used as input signal for Timer 2, ETR or as channel 1 (TIM2_CH1) if alternate function AF1 is assigned to this pin. Alternatively, this pin can be used to pass signals for SPI communication if alternate functions AF5 or AF6 are selected.

The complete code for the configuration of pin 15, port A, to serve as an input pin for signal ETR is put in a function “GPIOAinit_TIM2_ETR()” listed below.

```
void GPIOAinit_TIM2_ETR(void)   {
GPIO_InitTypeDef          GPIO_InitStructure;                                          // 2

  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);                                // 4

  GPIO_PinAFConfig(GPIOA, GPIO_PinSource15, GPIO_AF_TIM2);          // PA15:TIM2_Ch1/ETR  // 6

  GPIO_InitStructure.GPIO_Pin          = GPIO_Pin_15;                                  // 8
  GPIO_InitStructure.GPIO_Mode         = GPIO_Mode_AF;                                 // 9
  GPIO_Init(GPIOA, &GPIO_InitStructure);                                               // 10
}
```

The structure of this function is common for a configuration of a pin. The second line declares the data structure, and the fourth line enables the clock for port A. The sixth line allows pin 15 (‘GPIO_PinSource15’) of port A (‘GPIOA’) to be used as alternative function for Timer 2, therefore the ETR or CH1. All it takes is to call the function “GPIO_PinAFConfig()” with these parameters. The pin is not yet put into the alternate function mode, only one of all possible alternate functions has been selected. The rest of the function changes the mode for the selected pin from normal to alternate function. The data structure is initialized in lines 8 and 9, note the ‘GPIO_Mode_AF’ in line nine. Finally, the pin gets initialized by the call of function “GPIO_Init()” in line 10.

Now for the configuration of Timer 2 to accept the signal from port A and count it. All code is packed into a function “TIM2init_counter()” listed below.

```
void TIM2init_counter(void) {       // counting from ETR
TIM_TimeBaseInitTypeDef   TIM_TimeBaseInitStructure;                                 // 2

  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2,  ENABLE);                              // 4

  TIM_TimeBaseInitStructure.TIM_Prescaler = 0;                                       // 6
  TIM_TimeBaseInitStructure.TIM_CounterMode = TIM_CounterMode_Up;                    // 7
  TIM_TimeBaseInitStructure.TIM_Period = 100000;                                     // 8
  TIM_TimeBaseInitStructure.TIM_ClockDivision = TIM_CKD_DIV1;                        // 9
```

```
   TIM_TimeBaseInit(TIM2, &TIM_TimeBaseInitStructure);                                    // 10

   TIM_ETRClockMode2Config(TIM2, TIM_ExtTRGPSC_OFF, TIM_ExtTRGPolarity_NonInverted, 0x00);      // 12

   TIM_Cmd(TIM2, ENABLE);                                                                 // 14
}
```

The configuration of Timer 2 follows the standard procedure briefly described in the source file “stm32f4xx_tim.c”; first the clock for the timer is enabled in line 4, second the timer is configured for counting direction, period and scaling in line 10; a CMSIS function call is used for easy configuration. Last the source of the signal to be counted is specified in line 12. A single data structure is needed for the second step, and it is declared at the beginning of the function, line 2.

The second step needs further discussion. The function “TIM_TimBaseInit()” is given in the source file “stm32f4xx_tim.c”, line 288, and the data structure to go with it is given in the header file “stm32f4xx_tim.h”, line 55 to 78.

- The first member ‘.TIM_Prescaler’ gives the prescaler value to be implemented in the block PSC in front of the counter CNT, see Fig. 9.2. The frequency of the input signal to the prescaler is simply divided by the value written as the prescaler value, or, in other words, if the value is 4 then only every fourth pulse will be passed to the input of the counter CNT. The prescaler value can be set to anything between 0 and 65535; since we want to count all pulses this member is initialized to 0.
- The second member ‘TIM_CounterMode’ defines the order of counting, is our case up-counting is selected by initializing this member with option ‘TIM_CounterMode_UP’. Other options are listed in the header file, lines 317 to 321.
- The third member ‘TIM_Period’ defines the content of the reload register; the content of the counter will reset back to 0 once the number in the reload register is reached. When a clock signal is connected to the input of the counter, this value specifies the period of counting. In out example we want to count up to a reasonable number before the content of the counter is reset back to 0, so this member is initialized to 100000.
- The fourth member ‘TIM_ClockDivision’ configures another divider of clock signal to be 1, ½ or ¼. In our case the divider is skipped by initializing this member to 0 (‘TIM_CKD_DIV1’).
- There is also a fifth member ‘TIM_RepetitionCounter’, but this option is not used in this experiment and the initialization of this member is not needed.

The third step to select the clock source also requires some additional discussion; the details are given in reference manual RM0090, chapter 18.3.3, pg. 587. Since we use ETR pin as a source of the signal to count, this requires the use of “External Clock mode 2”. There is a function in the CMSIS library for every possible clock source, and the one for the use of ETR signal as a clock is named “TIM_ETRClockMode2Config()”. The function requires following arguments.

- First argument is the name of the counter to be configured, TIM2 in this case.
- Second argument configures yet another prescaler inserted in the path of ETR signal. This prescaler can divide the frequency of the input signal by 2, 4 or 8, and it can also be disabled as in our case, where this parameter is set to ‘TIM_ExtTRGPSC_OFF’.
- The third argument configures the polarity of the ETR signal that is used to increment the counter, in this case it is set to true (‘TIM_ExtTRGPolarity_NonInverted’).
- The last argument configures the filter inserted into the path of ETR signal. Here we do not to filter the signal, and the value is set to 0 (it can range from 0 to 15), see the reference manual RM0090 for details.

The demo program is written to initialize the port and timer TIM2, and then enter the infinite loop where it periodically reads the content of the timer and writes it on the LCD. The complete program is given in the listing below.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_tim.c"                                                    // 4
#include "LCD2x16.c"

int main ()  {
  GPIOAinit_TIM2_ETR();               // GPIO clock enable, digital pin definitions      // 8
  TIM2init_counter();                 // Timer 2 as counter: ETR input                  // 9
  LCD_init();                         // Init LCD                                       // 10
  LCD_string("TIM2 counter ETR", 0);                  // write title                    // 11

  while (1) {
    LCD_uInt32(TIM2->CNT, 0x40, 0x01);                // show counts                    // 14
    for (int i = 0; i < 1000000; i++)   {};           // waste time                     // 15
  };
}
```

The program starts with a set of include statements, note the fourth line to include the source file “stm32f4xx_tim.c”. The main function starts with three calls to initialize port A, Timer2 and LCD screen. The two functions listed above should be included in order for the program to compile. The last line before entering the infinite loop (11) writes some introductory text to the LCD screen.

Within the infinite loop the microcontroller simply reads the content of the counter register in timer TIM2 (32 bits), and writes the value on the LCD. Please note that the content of the counter could also be read by CMSIS function for better portability, but is not implemented here. The execution of the loop is slowed down by inclusion of an empty for loop.

10.Interrupts and ports

The interrupt mechanism is very important for an effective use of processor time and resources, and is implemented in every microcontroller. A simple example of the use of interrupts is given here.

10.1. The hardware – NVIC, the Nested Vectored Interrupt Controller

The processor executes a program line after line as it was written by the programmer. Jumps from one part of the program to another part are possible, and are preprogrammed or/and used to follow decisions based on the values of variables; they are predictable at the time of writing of the program.

Many times an important event outside of the processor requires immediate attention and action of the processor while it is executing a program. The situation where the execution of the program is suspended and the processor is forced to take care of the important event is called an interrupt. Such situation is initiated by an electrical signal called interrupt request. The interrupt request can be issued by a hardware built into the microcontroller, but can also be issued by an external hardware connected to a pin of a port. This example will show how to program a microcontroller to respond to the interrupt request caused by pressing a pushbutton connected to port E, pin 3.

A special block of hardware is built into the microcontroller in order to release the processor from complex tasks involved in handling interrupt requests. This hardware is called interrupt controller, and is capable of receiving an interrupt request signal, interpreting it as a justified reason to stop the processor from executing the regular program and actually forcing the processor to jump to and execute a special piece of code called interrupt function. It is the task of the processor to return to the regular program after the execution of the interrupt function and continue its execution as if nothing had happened.

In STM32F4xx series microcontrollers the interrupt controller is called Nested Vectored Interrupt Controller (NVIC, reference manual RM0090, chapter 12, page 369, and programming manual PM0214, chapters 2.3 and 4.3.7). The controller NVIC can handle interrupt requests caused by most hardware (like DMA requests, interrupt requests from communication channels USART, CAN, I2C, and timers)

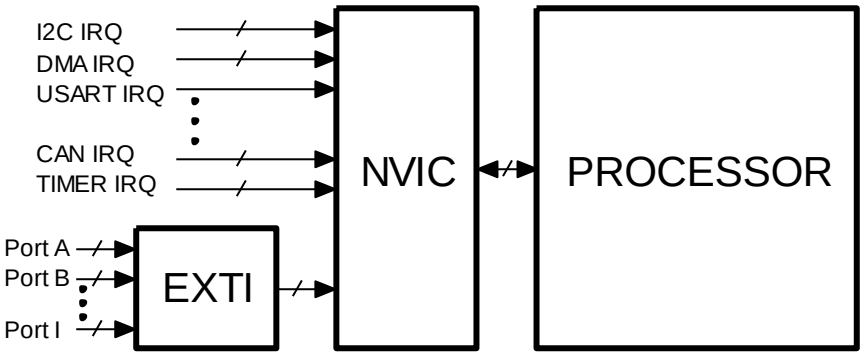


Figure 10.1: The chain involved in the processing of interrupt requests - simplified

built into the microcontroller alone, and also receives interrupt requests from ports through an additional hardware called External interrupt/event Controller (EXTI). A simplified block diagram for the processing of interrupt request signals in shown in Fig. 10.1.

In order to utilize the interrupt capabilities of the microcontroller the following must be fulfilled:

- The interrupt function must be prepared. This means that the programmer must know what to do in case of an interrupt request, and must write the code (interrupt function) to be executed due to the interrupt. The interrupt function must be written in a special way not to harm the execution of the regular program and the compiler must be notified that this is the function to be executed in case of a certain interrupt request on a certain event. The interrupt function cannot receive or return any arguments as normal functions do; the arguments used within the interrupt function must be declared as global at the beginning of the regular program.
- The interrupt controller NVIC must be configured and enabled. Since changing of the settings within the interrupt controller is a delicate task and the registers involved into configuration of this controller can be accessed only from a privileged state of the processor, a set of functions to change/configure the controller NVIC is included in a CMSIS library within the compiler package (“core_cm3.h”, “misc.c”, “misc.h”).
- The controller EXTI must be configured and enabled to sort-out the signals at ports and interpret them as interrupt requests.

Once the controller NVIC receives the interrupt request it validates it and notifies the processor. The processor then requests the code of the interrupt request source and uses it to find the pointer to the corresponding interrupt function from the interrupt vector table. The table is located in the memory of the microcontroller, and has more than 80 entries for vectors. The complete list of vectors is given in Table 62, RM0090, pg. 368. The interrupt requests have priorities, some are more important than others. The default priority for interrupts is given in column 2, but can be changed from software. From this table one can see that there are seven vectors reserved for interrupt requests from ports; these vectors are named EXTIx, where x stands for the number (EXTI0 as table entry number 6, priority 13, …). These seven vectors can serve 16 interrupt requests issued through ports; some vectors are shared by more than one interrupt request.

The vector table itself as used by the compiler can be seen in assembly language source file “startup_stm32f4xx.s” from line 57 on. Individual vectors to point to corresponding interrupt functions are placed into this table during the compilation process, and the functions to be executed on individual interrupt requests must have the same names as listed in this table. As an example, the interrupt function to be executed on external interrupt EXTI3 must have the name “EXTI3_IRQHandler”. The required names of interrupt functions for other interrupt request can be obtained from this vector table.

As stated the controller NVIC can be handled only in privileged mode, and a set of functions to change registers of the controller has been made available in CMSIS library to ease the setting of the controller. These are:

- NVIC_EnableIRQ (IRQ#): Enables the handling for the given interrupt request (IRQ#) as specified in Table 62, RM0090, page 3768, first column. The number IRQ# can also be replaced by a human-friendly name defined in file »stm32f4xx.h«, line 160 and on.
- NVIC_DisableIRQ (IRQ#): Disables the handling for the given interrupt request (IRQ#).
- NVIC_SetPendingIRQ (IRQ#): Sets the status of an interrupt request as pending.
- NVIC_ClearPendingIRQ (IRQ#): Clears the status of an interrupt request from being pending.

- NVIC_GetPendingIRQ (IRQ#): Inquires about the pending status of a given interrupt request (IRQ#) and returns a non-zero value if IRQ# is pending.
- NVIC_SetPriority (IRQ#, priority#): Sets the priority for the given interrupt request (IRQ#) with configurable priority level to value stated by priority#.
- NVIC_GetPriority (IRQ#): Reads the priority for the given interrupt request with configurable priority level.

The controller EXTI handles the processing of interrupt request signals from all pins of ports (144 of them, 16 pins by 8 ports in this microcontroller family) and concentrates the requests into 16 (23 including some extra interrupt request sources, see RM0090, chapter 12, for details) lines to be passed on to controller NVIC. The controller EXTI consists of 16 multiplexors and additional hardware, see fig. 10.2. (only three multiplexors are shown). Multiplexors concentrate interrupt request signals from pins into fewer lines EXTIx. Signals from pins 0 from all ports enter a multiplexor at the left-top, and one of them can be used as an EXTI0 interrupt request signal. Each multiplexer is controlled by four bits of register contained in system configuration hardware, which must be clocked in order to work. Signals from pins 1 from all ports enter the next multiplexer, and one of them can be used as an EXTI1 interrupt request signal. The same pattern repeats for other pins.

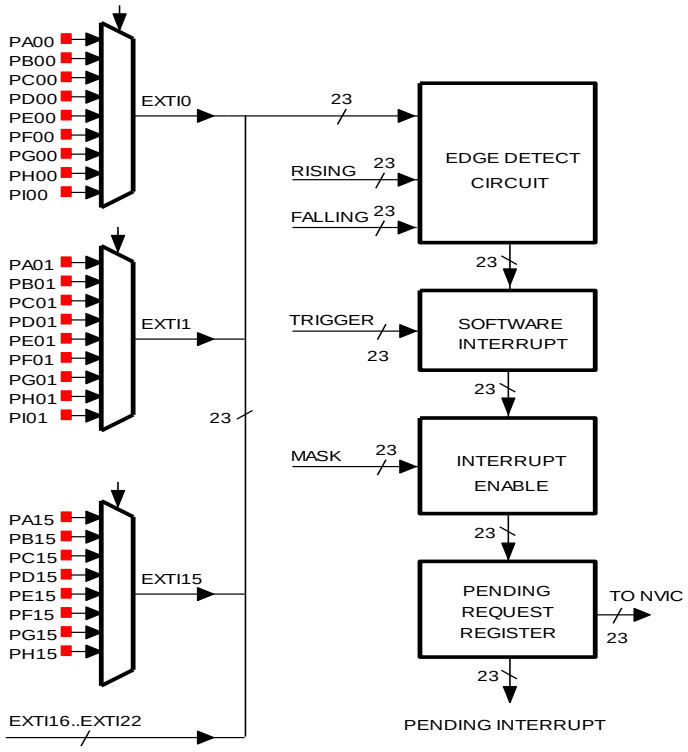


Figure 10.2: The controller EXTI, red dots represent input pins

The resulting bus EXTI is 23-bits wide and enters a circuit for edge detection; an interrupt can be issued on rising and/or falling edge of a signal, and the appropriate edge for each of the 23 signals is selected individually by setting signals 'RISING' and 'FALLING'.

An interrupt request can be also triggered by software, as indicated by the next block in the chain; setting a bit TRIGGER initiates the interrupt processing as if it would be initiated by hardware. However, not all interrupt request signals can actually trigger the interrupt. A mask to enable individual interrupt request signals in implemented in the next block. Only enabled interrupt requests can pass to the last block to be memorized before entering the interrupt controller NVIC.

Interrupt requests are prioritized; when processor is executing a high priority interrupt function other, lower priority requests, must wait. Those are called "pending interrupt requests". The software can check about such pending requests by investigating the 'PENDING INTERRUPT' signals exiting the last block in the chain. The same register also memorizes an interrupt request, and this register must be cleared from the interrupt function to avoid the repeated execution of the same interrupt function for one interrupt request.

10.2. The software to test the interrupt issued by pressing a pushbutton S370

A short program is presented to confirm the operation of interrupt hardware. In the main program the content of a variable is be continuously written to the LCD screen. When a user presses a pushbutton, a variable is incremented within the interrupt function. The pushbutton is connected to port E, pin 3.

The use of the interrupt mechanism is, as far as programming is concerned, split into two parts: function for the configuration of the interrupt hardware, and the actual interrupt function.

- The function for the configuration of the hardware is named “IRQinit_PE3()” in this example and is given below.

```
void IRQinit_PE3 (void) {
EXTI_InitTypeDef        EXTI_InitStructure;

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);                              // 4
  SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOE, EXTI_PinSource3);                       // 5

  EXTI_InitStructure.EXTI_Line    = EXTI_Line3;                                       // 7
  EXTI_InitStructure.EXTI_Mode    = EXTI_Mode_Interrupt;                              // 8
  EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;                              // 9
  EXTI_InitStructure.EXTI_LineCmd = ENABLE;                                           // 10
  EXTI_Init(&EXTI_InitStructure);                                                     // 11

  NVIC_EnableIRQ(EXTI3_IRQn);                 // Enable IRQ for ext. signals, lines 3         // 13
}
```

The registers holding the configuration for multiplexors are configured first, and those registers are located in microcontroller block ‘SYSCFG’; this block needs a clock signal first; it is enabled in the fourth line of the function. Once the clock is present the multiplexor for signal EXTI3 (for pins 3, not shown in Fig. 10.2) can be configured to pass the signal from Port E. This is done by function call in the fifth line of the function, where ‘EXTI_PortSourceGPIOE’ is selects the ‘PE’ input of the multiplexor for EXTI3. This finishes the configuration of the multiplexor.

Other four blocks are best configured using a CMSIS function “EXTI_Init()”, described in the source file “stm32f4xx_exti.c”, and having its data structures and definitions described in the header file “stm32f4xx.exti.h”. Lines 7 to 10 are used to initialize the members of the data structure ‘EXTI_InitStructure’, which is declared in the second line of the function. The pointer to this data structure is the argument of the function call in the 11th line:

- The first member ‘.EXTI_Line’ selects one of the EXTIx signals to be affected by the call to the “EXTI_Init()” function.
- The second member ‘EXTI_Mode’ selects the configuration of interrupt or event structure, see the RM0090 for details.
- The third member ‘EXTI_Trigger’ selects the edge to trigger the interrupt request. It can be either ‘EXTI_Trigger_Rising’, ‘EXTI_Trigger_Falling’ or ‘EXTI_Trigger_Rising_Falling’.
- The last member ‘.LineCmd’ sets the mask to be used and actually enables or disables the EXTI line.

The NVIC controller is configured next. Since the only interrupt request is coming from the pushbutton, there is no need to set its priority, and we only need to enable it. This is done in line 13 with a function call “NVIC_EnableIRQ()” with an argument ‘EXTI3>_IRQn’.

The interrupt function to be executed on the press of a pushbutton is given in the listing bellow:

```
void EXTI3_IRQHandler (void)   {
  IRQcounter++;                              // do the actual work                          // 2
  EXTI_ClearITPendingBit(EXTI_Line3);                                                    // 3
  EXTI_ClearFlag(EXTI_Line3);                                                            // 4
}
```

The function must not receive or return any arguments, and it is declared as 'void' for both. Its name must be as defined in the vector table located in the file "startup_stm32f4xx.s". The body of the interrupt function starts in line 2, where the counter variable 'IRQcounter' is incremented. Remember: this must be a globally declared variable. Two bits in the last register from Fig. 10.2 must still be cleared to confirm the execution of the interrupt function for this interrupt request. The two bits are reset by calling two CMSIS functions at lines 3 and 4.

In order to use the pushbuttons the port E must be declared as input with pull down resistors at lines PE3 to PE7. A function "SWITCHinit()" to do this has been presented already, and must be included in the final listing, given below.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_exti.c"
#include "stm32f4xx_syscfg.c"
#include "LCD2x16.c"

int IRQcounter = 0;                          // declare & initialize IRQ counter            // 8

void main (void){

  LCD_init();                                // init LCD                                    // 12
  LCD_string("IRQs at PE3:", 0x00);          // display title string                        // 13

  SWITCHinit();                                                                          // 15
  IRQinit_PE3();                                                                         // 16

  while (1) {
    LCD_uInt16(IRQcounter,0x40,1);           // write IRQ count to LCD                      // 19
    for (int i = 0; i<100000; i++) {};       // waste some time                             // 20
  };
}
```

The listing commences with several include commands. Note the "stm32f4xx_exti.c" and "stm32f4xx_syscfg.c" for CMSIS function related to the newly used peripherals. A global variable 'IRQcounter' is declared and initialized in line 8. The main part of the program starts with a call to the initializing function for the LCD and by writing of the introductory string to the LCD, lines 12 and 13. Next port E is initialized for pushbuttons, and interrupts are initialized and enabled. This part is followed by an infinite 'while' loop, where the content of the variable 'IRQcounter' is periodically written to the LCD in line 19. A 'for' loop is added to slow down the execution of the program.

11.Interrupts and Timer

The interrupt mechanism is very important for a programmer, and is implemented in every microcontroller. An interrupt request can come from many sources, here it is generated periodically by Timer 5. Such approach can be used for instance for periodical sampling or generating of a signal, or for periodical switching between tasks in a multitasking system.

11.1. The hardware – NVIC, the Nested Vectored Interrupt Controller

The discussion on interrupt processing hardware was given in chapter 10 (Interrupts and ports). Here an example of the use of timer and interrupt processing will demonstrate how to periodically interrupt the execution of the regular program in such a way that the processor can do some work at precisely defined time intervals. In our case the processor will generate two consecutive pulses at port E, bit 8, for every interrupt request.

The block diagram representing the behavior of the microcontroller for this experiment is given in Fig. 11.1. The timer TIM5 is clocked by a signal CLK with frequency of 84MHz, and is configured to make periodical pulses, the period T being 1ms. Each of these pulses is passed to Nested Vectored interrupt Controller NVIC to issue an interrupt to the processor, and processor in turn makes a double pulse at port E, pin 8.

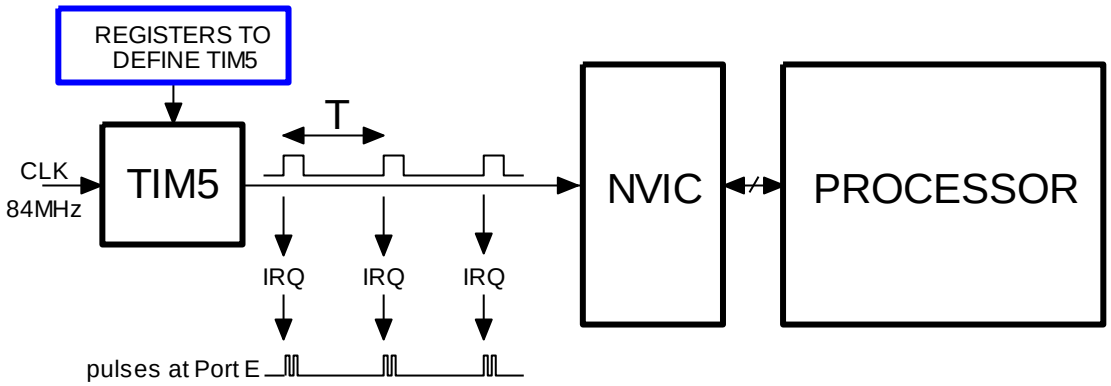


Figure 11.1: The chain used to implement periodic interrupt requests - simplified

The timer TIM5 has the same capabilities and block structure as previously described timer TIM2; it consist of a 32-bit counter, several multiplexors to select the clock source and additional hardware to control the counting. The clock signal CLK is one of the internal clocks of the microcontroller, and is connected to the input of the counter through the set of multiplexors by default.

The timer TIM5 counts up to a number stored in its auto-reload register and then resets back to zero; the return to zero is called an update event, and can be configured to trigger an interrupt. Setting the content of the auto-reload register to 84000 and using the clock signal with the frequency of 84 MHz assures the time interval T to be 1 ms. The controller NVIC must be configured to receive pulses (update events) from the timer TIM5.

11.2. The software – the making of double pulses once per millisecond

The complete configuration for the timer TIM5 and the NVIC can be put into a function and named "TIM5init_TimeBase_ReloadIRQ()". This function is called prior to the use of timer interrupts, and takes the time interval between consecutive interrupts as an argument; note that integer variable is 32 bits in length. The listing of the function is given below.

```
void TIM5init_TimeBase_ReloadIRQ (int interval)  {
TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStructure;                                  // 2

  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5,  ENABLE);                             // 4

  TIM_TimeBaseInitStructure.TIM_Prescaler = 0;                                      // 6
  TIM_TimeBaseInitStructure.TIM_CounterMode = TIM_CounterMode_Up;                   // 7
  TIM_TimeBaseInitStructure.TIM_Period = interval;                                  // 8
  TIM_TimeBaseInitStructure.TIM_ClockDivision = TIM_CKD_DIV1;                       // 9
  TIM_TimeBaseInitStructure.TIM_RepetitionCounter = 0;                              // 10
  TIM_TimeBaseInit(TIM5, &TIM_TimeBaseInitStructure);                               // 11

  NVIC_EnableIRQ(TIM5_IRQn);                 // Enable IRQ for TIM5 in NVIC         // 13
  TIM_ITConfig(TIM5, TIM_IT_Update, ENABLE); // Enable IRQ on update for Timer5     // 14

  TIM_Cmd(TIM5, ENABLE);                                                            // 16
}
```

The timer is configured using a CMSIS function "TIM_TimeBaseInit()", given in the source file "stm32f4xx_tim.c". The same function was already used in chapter on counting pulses using the timer TIM2. This function requires the use of data structure named 'TIM_TimeBaseInitStructure' described in the header file "stm32f4xx_tim.h", lines 55 to 78. The data structure is declared in the second line of the function, and the clock for the timer TIM5 is enabled in line 4. The members of the data structure are initialized in lines 6 to 10 as follows.

- The first member '.TIM_Prescaler' is set to 0, since we do not want to reduce the frequency of the clock signal.
- The second member '.TIM_CounterMode' is initialized to 'TIM_CounterMode_Up' to make the counter advance on clock pulse.
- The third member '.Tim_Period' is the number to be stored into the auto-reload register; we could use value 84000 in this example to define the time interval T to 1ms. However, in order to make this function more flexible the value is not fixed, but is sent to this function as an argument.
- The fourth member '.TIM_ClockDivision' is set to 0 since we do not want to reduce the frequency of the clock signal.
- The last member '.TIM_RepetitionCounter' is irrelevant for this example, but nevertheless initialized to 0.

The timer TIM5 is then configured by calling the function "TIM_TimeBaseInit()" in line 11; the timer has not been started yet. Next comes the configuration of the controller NVIC, which must be enabled to receive interrupt request pulses from timer TIM5. This is done by a call in line 13. The last step in configuration is to allow timer TIM5 to use the update events as a source of the interrupt request pulses, and this gets configured by a call to function "TIM_ITConfig()" in line 14. The full description of this function is given in the source file, lines 2372 to 2389, and possible names of events within the timer to be used as interrupt requests are defined in the header file, lines 553 to 560.

The last thing to do is to enable timer TIM5 to commence the counting, this is done in line 16.

The interrupt function to be executed on interrupt must be prepared and written as a stand-alone function taking and returning no arguments. In our case four statements are needed to toggle port E, bit 8, high-low-high-low. Additionally, the update event of timer TIM5 toggles a flag stored in status register of timer TIM5, and must be cleared within the interrupt function to prevent immediate repetition of the same interrupt request. The complete listing of the function is given below.

```
void TIM5_IRQHandler(void)       {
int i;
  TIM_ClearITPendingBit(TIM5, TIM_IT_Update); // clear interrupt flag                    // 3
  GPIOE->ODR |=  BIT_8;        for (i=0; i<10; i++) {};                                 // 4
  GPIOE->ODR &= ~BIT_8;        for (i=0; i<10; i++) {};                                 // 5
  GPIOE->ODR |=  BIT_8;        for (i=0; i<10; i++) {};                                 // 6
  GPIOE->ODR &= ~BIT_8;                                                                 // 7
}
```

Note the name of the interrupt function “TIM5_IRQHandler”, it must be the same as specified in the interrupt vector table, file “startup_stm32f4xx.s”, line 133.

The order of events is then as follows. When the timer TIM5 content reaches the predefined value stored in the reload register the content of the counter register CNT returns to zero triggering a reload event, and the counting continues from zero on. The reload event turns on a flag in status register of timer TIM5, and is simultaneously passed to the controller NVIC as a signal to trigger an interrupt request. Since the controller NVIC is enabled to respond to this particular interrupt request, it forces the processor to interrupt the execution of the regular program and starts executing the relevant interrupt function. Within the interrupt function the processor toggles the port E, bit 8, four times to make two consecutive pulses and clears the update event flag in the status register of timer TIM5 to acknowledge the execution of the interrupt function. After this the processor continues with the execution of the regular program as if nothing had happened.

The listing of the demo program is given below, it must be complemented with the two functions listed above and a function to initialize port E, pins 8 to 15 as outputs, see chapter 4 for details.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_tim.c"
#include "dd.h"

void main(void) {

  GPIOEinit ();                                                                         // 9
  TIM5init_TimeBase_ReloadIRQ (84000);           // 84000 == 1ms                       // 10

  while (1) {};
}
```

The listing starts with the include statements, here all relevant source files are added to the user program. The main section of the listing commences with a call to initialize port E and a call to initialize timer TIM5 and the controller NVIC, and then enters an infinite loop where the processor simply waits for an interrupt. On interrupt the processor executes the interrupt function “TIM_IRQHandler()”, and then returns to the infinite loop.

12. Periodical interrupts and ADC/DAC

The knowledge from previous chapters will be used here to prepare a program which can periodically start a conversion at the ADC, wait for the result and pass the result to the DAC.

12.1. The hardware – idea on sampling and generating

The previously given demonstration program for ADC shows the initialization needed to use two of the built-in ADCs to sample two analog input signals. The results were displayed at the LCD. Here the results will be passed to two DACs generating a replica of the input signal, but quantized in time and value. The picture in Fig. 12.1 shows one of the input signals and the corresponding quantized version at the output from a DAC. Such program could also be prepared by combining former examples on the use of ADC and DAC, and inserting the relevant code for start conversion at the ADC, waiting of the conversion result, and sending the result to DAC in a loop. An additional empty loop could define the time delay between two consecutive iterations of the loop and therefore the sampling frequency. However, using the software to define the time interval is wasting of the processor time, and should be avoided. A additional and much better reason not to use this technique will become evident in the following chapters: we usually want to process the signal before passing to the DAC, and the processing time may vary on the nature of processing.

Figure 12.1: The quantized-in-time version of the input signal

It has been demonstrated in chapter 11 that time intervals can be defined using a timer, and that the reload events of a timer can trigger interrupts. It is therefore only natural to define time intervals between two successive conversions by a timer, and therefore use interrupt function to start the conversion at ADC by issuing the pulse Start Conversion SC, wait for the result, and then pass the result to DAC, as depicted in Fig. 12.2.

12.2. The software – implementation of the idea

The listing of the program to implement the idea above is simple and based on previous three chapters 7, 8, and 11. The program from chapter 11 defines time intervals. A software command is used to start the conversion as demonstrated in program from chapter 7, but this time the command

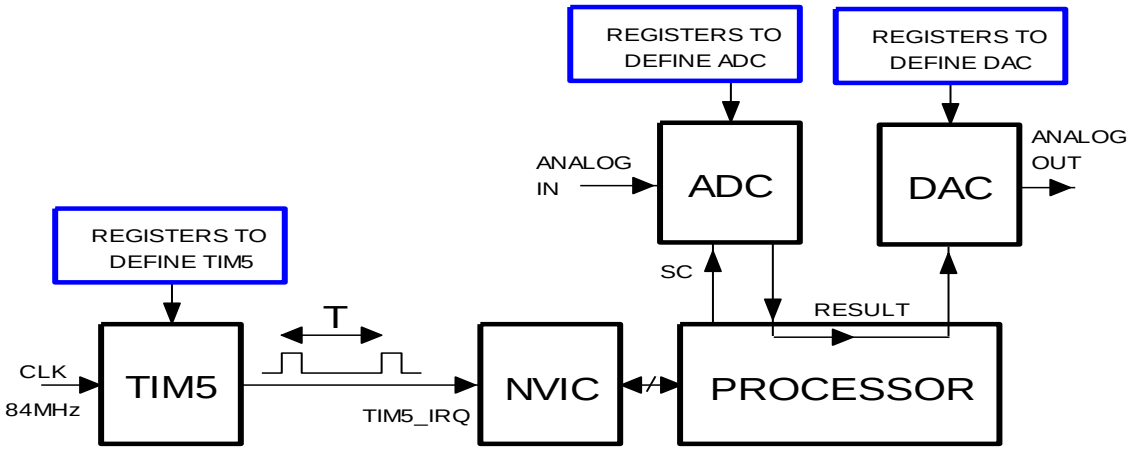


Figure 12.2: The natural operation for periodical sampling and generating of analog signals

is issued from the interrupt function, and therefore issued periodically. The result of conversion is passed to the DAC as demonstrated in chapter 8, still inside the interrupt function. Three hardware blocks require configuration, these are the ADC, the DAC, and the timer. All configuration functions ("ADCinit_SoftTrigger()", "DACinit()", and "TIM5init_TimeBase_ReloadIRQ()") are copied from the relevant chapters. The listing of the program is given bellow.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_adc.c"
#include "stm32f4xx_dac.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_tim.c"

int main ()  {

  ADCinit_SoftTrigger();                                                    // 10
  DACinit();                                                                // 11
  TIM5init_TimeBase_ReloadIRQ(840);                  // 840 == 10us         // 12

  while (1) {                                        // endless loop
  };
}

void TIM5_IRQHandler(void)       {                                          // 18
  TIM_ClearITPendingBit(TIM5, TIM_IT_Update);        // clear interrupt flag
  DAC->DHR12R1 = ADC1->DR;                                                  // 20
  DAC->DHR12R2 = ADC2->DR;
  ADC_SoftwareStartConv(ADC1);                                              // 22
}
```

The program starts with a set of include statements to gather all CMSIS functions into the user source file. The main part of the program commences with three calls to configuration functions, note that here the argument to define the time interval between two consecutive timer interrupts is reduced to 840 (10μs). The processor enters the infinite empty loop after the configuration, and simply waits for the interrupt request.

The interrupt function starts at line 18. The first thing to do here is to clear the flag in timer TIM5 to prevent a repeated execution of the interrupt function. In order to expedite the execution of the function the order of actions as stated above is reversed. Line 20 copies the current content of the ADC1 into the DAC1, and the line 21 does the same for ADC2 and DAC2. Next in line 22 the ADC

conversion is started. It will be finished well before the next repetition of the interrupt function (within 10μs), and the results of this conversion will be waiting then to be copied to DACs.

Such implementation is straightforward, but may not be the best. The digital signal processing theory states that it is very important to take samples at exact time intervals, or the signal to noise ratio will be impaired. The exact period is not guaranteed in the former program since the processor can start executing the interrupt function only after it finishes the execution of the current instruction, and instructions in complex programs can take different time to execute. Furthermore, the execution of the interrupt function for timer TIM5 itself might be delayed in more complex programs utilizing more than one interrupt function if the interrupt request for timer TIM5 is assigned low priority.

12.3. The software – a better implementation

To avoid these problems we should trigger the start of conversion SC directly by a timer TIM5 without the intervention of the software, and use interrupt function only to copy the result of conversion to the DAC. Such interrupt function shall be initiated by the end of conversion signal EOC coming from the ADC, which can also be used to trigger an interrupt at ADC_IRQ as depicted in Fig. 12.3.

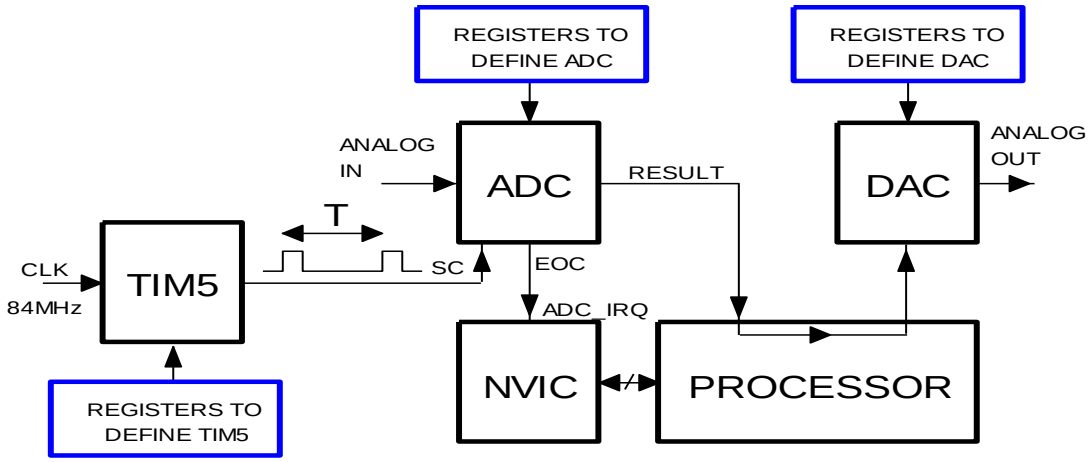


Figure 12.3: The start of conversion can also be triggered by timer TIM5, and the end of conversion signal EOC is used to trigger an interrupt. The interrupt function moves the result of conversion from ADC to DAC.

Few things need to be changed in the configuration section for the ADC, the timer and the interrupt function itself. The configuration section for the DAC remains the same for now.

We start with the configuration of the ADC, the new function is called “ADCinit_T5_CC1_IRQ()”, and is listed below.

```
void ADCinit_T5_CC1_IRQ (void)      {
ADC_InitTypeDef         ADC_InitStructure;
GPIO_InitTypeDef        GPIO_InitStructure;
ADC_CommonInitTypeDef   ADC_CommonInitStructure;

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1  | RCC_APB2Periph_ADC2,  ENABLE);
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA | RCC_AHB1Periph_GPIOB, ENABLE);

  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_1 | GPIO_Pin_2;
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AN;
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;
  GPIO_Init(GPIOA, &GPIO_InitStructure);

  ADC_CommonInitStructure.ADC_Mode             = ADC_DualMode_RegSimult;
```

```
  ADC_CommonInitStructure.ADC_Prescaler        = ADC_Prescaler_Div2;
  ADC_CommonInitStructure.ADC_DMAAccessMode    = ADC_DMAAccessMode_Disabled;
  ADC_CommonInit(&ADC_CommonInitStructure);

  ADC_InitStructure.ADC_Resolution             = ADC_Resolution_12b;
  ADC_InitStructure.ADC_ScanConvMode           = DISABLE;
  ADC_InitStructure.ADC_ContinuousConvMode     = DISABLE;
  ADC_InitStructure.ADC_ExternalTrigConvEdge   = ADC_ExternalTrigConvEdge_Rising;          // 22
  ADC_InitStructure.ADC_ExternalTrigConv       = ADC_ExternalTrigConv_T5_CC1;              // 23
  ADC_InitStructure.ADC_DataAlign              = ADC_DataAlign_Right;
  ADC_InitStructure.ADC_NbrOfConversion        = 1;
  ADC_Init(ADC1, &ADC_InitStructure);
  ADC_RegularChannelConfig(ADC1, ADC_Channel_1, 1, ADC_SampleTime_3Cycles);

  ADC_Init(ADC2, &ADC_InitStructure);
  ADC_RegularChannelConfig(ADC2, ADC_Channel_2, 1, ADC_SampleTime_3Cycles);

  ADC_Cmd(ADC1, ENABLE);
  ADC_Cmd(ADC2, ENABLE);

  NVIC_EnableIRQ(ADC_IRQn);                // Enable IRQ for ADC in NVIC                 // 35
  ADC_ITConfig(ADC1, ADC_IT_EOC, ENABLE);  // Enable IRQ generation in ADC                // 36
}
```

The differences start in line 22. We need to use a hardware signal to trigger the start of conversion, and this is configured with initializing the member '.ADC_ExternalTrigConvEdge' to accept the rising edge of an external signal. This line basically enables the output of a multiplexor below the ADC, Fig. 8.1 to start the conversion with its rising edge. There are three inputs to the multiplexor related with timer TIM5, all of them are outputs from Capture and Compare blocks within timer TIM5. They are called CH1, CH1 and CH3, see Fig. 9.2 for reference. Whichever we use we have to configure timer TIM5 to implement capture and compare function, and here we opt to use channel CH1 as the source of start conversion pulses by using the keyword 'ADC_ExternalTrigConv_T5_CC1' to initialize the member '.ADC_ExternalTrigConv'.

The second difference is the addition of statements to allow the end of conversion signal to be used as an interrupt request. Two steps are necessary.

- The first step is used to configure the controller NVIC. A call to function "NVIC_Enable()" with an argument 'ADC_IRQn' allows the controller NVIC to react on an interrupt request signal 'ADC_IRQ' from the ADC, line 35.
- The second step in line 36 configures the ADC to use its end of conversion signal EOC to issue an interrupt request to controller NVIC. There are other signals within the ADC that can be used to issue the interrupt request, a list of them is given in the header file "stm32f4xx_adc.h", lines 478 to 481, and can be seen in the reference manual RM0090, pg.387, where a detailed explanation is provided.

The new configuration function for the timer is dealt next, its listing is given below.

```
void TIM5init_TimeBase_CC1 (int interval)  {
TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStructure;
TIM_OCInitTypeDef       TIM_OCInitStructure;                                            // 3

  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5,  ENABLE);

  TIM_TimeBaseInitStructure.TIM_Prescaler = 0;
  TIM_TimeBaseInitStructure.TIM_CounterMode = TIM_CounterMode_Up;
```

```
  TIM_TimeBaseInitStructure.TIM_Period = interval;
  TIM_TimeBaseInitStructure.TIM_ClockDivision = TIM_CKD_DIV1;
  TIM_TimeBaseInitStructure.TIM_RepetitionCounter = 0;
  TIM_TimeBaseInit(TIM5, &TIM_TimeBaseInitStructure);

  TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;                              // 14
  TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;                  // 15
  TIM_OCInitStructure.TIM_OutputNState = TIM_OutputState_Disable;                // 16
  TIM_OCInitStructure.TIM_Pulse = 1;                                             // 17
  TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;                      // 18
  TIM_OCInitStructure.TIM_OCNPolarity = TIM_OCNPolarity_Low;                     // 19
  TIM_OCInitStructure.TIM_OCIdleState = TIM_OCIdleState_Reset;                   // 20
  TIM_OCInitStructure.TIM_OCNIdleState =  TIM_OCIdleState_Set;                   // 21
  TIM_OC1Init(TIM5, &TIM_OCInitStructure);                                       // 22

  TIM_Cmd(TIM5, ENABLE);                                                         // 24
}
```

The configuration is performed in two steps. The first step deals with counting, and is the same as given in chapter 11. The second step deals with capture and compare function needed to generate the SC pulses, and is new. The capture and compare function is described in detail in reference manual RM0090, chapter 18. A CMSIS function “TIM_OC1Init()” is used to ease the configuration of the output ‘OC1’ (equivalent functions are available for configuration of other outputs OC2 to OC4, and a brief description is given in the source file for timer), and this function requires the use of data structure “TIM_OCInitStructure”, which is declared in line 3 of the function. The members of the data structure are initialized in lines 14 to 21, and the function is called in line 22.

- The first member ‘.TIM_OCMode’ is initialized to a pulse width modulation (PWM) mode. Other options are listed in the header file “stm32f4xx_tim.h”, lines 240 to 245, and described in the reference manual RM0090.
- The second member ‘.TIM_OutputState’ is enabled to allow the generation of a pulse at the output from the timer OC1 to start the conversion.
- Next member ‘.TIM_OutputNState’; the negated version of the above signal can be generated as well, but is not needed and therefore disabled.
- Next member ‘.TIM_Pulse’ is initialized to 1 causing the signal OC1 to change its state when the content of the counter CNT equals 1.
- The fifth member ‘.TIM_OCPolarity’ is initialized to ‘TIM_OCPolarity_High’, causing the signal OC1 to change to high when the condition specified by the previous member is fulfilled.
- The sixth member ’.TIM_OCNPolarity’ is not needed here, but still initialized.
- The seventh member ‘.TIM_OCIdleState’ specifies the initial state of the OC1 after the reload event of the counter, and is configured to low (Reset) state.
- The last member ‘.TIM_OCNIdleState’ is not needed here, but still initialized.

The function call to enable timer interrupt requests, as given in chapter 11, is not needed here and has been removed. The configuration function terminates by enabling the counting in line 24.

The interrupt function deals with reading of the result of conversion from the ADC and writing of the same to DAC. Two lines suffice; they are given in the listing bellow.

```
void ADC_IRQHandler(void)      {
  DAC->DHR12R1 = ADC1->DR;
  DAC->DHR12R2 = ADC2->DR;
}
```

Please note the changed name of the interrupt function, since it is called due to a different interrupt request. Note also, that the flag to memorize the interrupt request from the ADC is stored in ADC block itself, and is automatically cleared by reading a result from the ADC, see the reference manual RM0090, section on ADC, for details. There is no need to clear this flag manually. Also, since conversions are now initiated by the timer, there is no need for software command to start the conversion.

The program to implement the timer triggered conversions is given in the listing below. It should be complemented with the inclusion of all functions described above.

```
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_adc.c"
#include "stm32f4xx_dac.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_tim.c"
#include "dd.h"

int main ()  {
  ADCinit_T5_CC1_IRQ();                                                    // 9
  DACinit();                                                               // 10
  TIM5init_TimeBase_CC1(840);           // 840 == 10us                     // 11

  while (1) {                           // endless loop
  };
}
```

The main part of the program calls the configuration functions and then enters an empty infinite loop, where it waits for interrupts from ADC. The timer TIM5 triggers the conversion at the ADC, and once the conversion is finished the end of conversion EOC signal from the ADC triggers the interrupt request to the NVIC and the processor. The processor breaks the execution of the empty loop and jumps to execute the interrupt function, where it reads the result of conversion (and simultaneously clears the flag memorizing the interrupt request) and transfers it to the DAC. After this the execution returns to the endless empty loop until the next interrupt.

The remark on periodic sampling and the urge to have equal time intervals between consecutive samples applies to periodic generating by the DAC as well. In order to reduce the harmonic distortion the DAC should move the data from the data holding register DHR to data output register DOR on signal derived from the same timer TIM5. This could be done by utilizing the multiplexor 'Trigger selectorx', see Fig. 7.1 for reference, and a change of the configuration of the DAC. However, this detail is omitted here.

13. The use of a circular buffer

A digital processing algorithms rely heavily on present and past samples of signals. The standard approach to memorize past values is the use of a circular buffer. Such structure will be described and its use will be demonstrated to generate a delayed version of a signal.

13.1. The background on storing past samples

A circular buffer is a reserved part of memory which serves as a temporary storage for data. It is organized in a special manner: the incoming data fills the reserved space until completely full, and then starts to fill the reserved space again from the beginning overwriting the previous content. It is the task of the software to make use of the stored data before it gets overwritten by new data, and the task of the programmer to reserve sufficient memory for the circular buffer.

It is customary to build a circular buffer using an array and a pointer into this array. However, the pointer must be bound to point into the array and never, under any circumstances, allow the access of data outside of the array since accessing the area outside of the array might have disastrous effects to the execution of the program. For instance the operation of writing into the circular buffer using a pointer, where the pointer points outside of the array, might corrupt the content of important memory locations, like registers or system variables. This must be prevented, and can be accomplished by two 'if' statements, see the simplified listing bellow.

```
int X[100], ptrX
….
  if (ptrX > 99) ptrX = 99;
  if (ptrX <  0) ptrX = 0;
…
```

In simple words: if the array has 100 elements, then the pointer should never be less than 0 or more then 99. These two 'if'statements should be inserted just before any use of the pointer 'ptrX' to access the content of the array 'X[ptrX]'. In a real processing this bounding of a pointer may look even more complex. The inclusion of these lines will slow down the execution of the program, and make the code less readable, so a better way is searched for.

Consider the use of an array with the length of 2^N, where N is a natural number. For the purpose of this example N equals 3, and the array has eight elements. Their indexes are from 0 to 7 in decimal notation, or from 00000000 to 00000111 in binary (8 bits notation). If we take any integer number named 'Ptr' written in binary, and use only the least significant three bits of this number as a pointer, these three bits point to one of the elements within this array. Therefore any integer number can be used as a valid pointer once we strip away all but the least significant three bits, and this can be done by a simple AND operation, where one of the arguments of the AND operation is the integer itself, and the other is the last available index within the array, 7 (00000111 binary) in our case. To be re-emphasized: this trick works only for arrays with the length of 2^N!

Now consider incrementing the variable 'Ptr' by one starting from 0, and its behavior as a pointer in the array. The pointer derived by AND-ing the variable 'Ptr' by 7 initially points to element 0 of the array, then element 1, then…. then element 7, and then element 0 again. The indexed elements form a circle, element 7 is followed by element 0.

Consider now decrementing the variable 'Ptr' by one, starting from, say, 2. The variable decrements from 00000010_2 to 0000001_2, then to 00000000_2, followed by 11111111_2, 11111110_2, and 11111101_2… Recall that values are written in two's complement when written as signed integers! The value 11111111_2 represents -1_{10}, and 11111110_2 represents -2_{10}. Taking the least significant three bits of the variable 'Ptr' again keeps the derived pointer within the bounds of the array, and selects elements 2, 1, 0, 7, 6, … and the element 0 is followed by the element 7, then 6…, and the indexed elements again form a circle, hence the array used in this way can be called a circular buffer.

The trick relies on a simple logic operation AND, and is effective for buffers with the length of 2^N elements. Other lengths need more complex bounding of the pointer, and are best avoided.

13.2. The implementation of a circular buffer

We shall implement a circular buffer to delay the generation of a signal. The program is based on the one derived in chapter 12, and the initialization of the hardware is identical and will not be re-discussed here. Both ADCs are used to periodically sample two input signals, and results from both ADS are stored in a circular buffers named 'X1[]' and 'X2[]'. One of the DACs is filled with the current result of the conversion, and the other is filled with whatever the result was 10 sampling intervals before. The listing is given below, the initialization functions are to be copied from chapter 12.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_adc.c"
#include "stm32f4xx_dac.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_tim.c"
#include "dd.h"

short int X1[1024], X2[1024], ptrX = 0;

int main ()  {
  ADCinit_T5_CC1_IRQ();
  DACinit();
  TIM5init_TimeBase_CC1(840);           // 840 == 100us

  while (1) {                           // endless loop
  };
}

void ADC_IRQHandler(void)       //
{
  ptrX++; ptrX &= 1023;
  X1[ptrX] = ADC1->DR;
  X2[ptrX] = ADC2->DR;
  DAC->DHR12R1 = X1[ptrX];
  DAC->DHR12R2 = X1[(ptrX-10) & 1023];
}
```

The distinctive details are:

- Two arrays named ‘X1[1024]’ and ‘X2[1024]’ are declared as global, they consists of 1024 elements (2^{10}) each, their indexes ranging from 0 to 1023. A pointer named ‘ptrX’ is declared as global and initialized to 0. Note that all variables used in an interrupt function must be declared as global.
- The interrupt function starts with a statement to calculate new value of the pointer in array. Variable ‘ptrX’ is first incremented, and then bound to stay within the range from 0 to 1023 forming a current pointer. Two results from ADCs are stored into arrays at current pointer. The content of current element of the array is copied into the first DAC. The content of the element which was stored into the array 10 sampling time intervals before is to be copied into the second DAC. The pointer to this element is calculated by subtracting 10 from a current pointer and bounding the value of the derived pointer by AND-ing it with the index of the last valid element (1023_{10} or 0x3ff in hex).

14. Serial communication – RS232

The common serial interface can be used to transfer data and commands between microcontrollers or a personal computer and a microcontroller. Here the configuration of the microcontroller to perform serial communication will be demonstrated.

14.1. The protocol RS232 and signals

A popular way to transfer commands or data between a personal computer and a microcontroller is the use of standard serial interface, like the one described by protocols RS232 (older stuff) or USB (newer, more capable). This chapter is devoted to communication conforming to RS232 protocol, the hardware for such interface is provided inside microcontroller STM32F407VG. An example will be presented showing the processing of commands received through RS232 interface, and sending of a string of numbers using the same interface.

The protocol RS232 defines signals used in communication, and properties of the hardware to transfer signals between devices. There are two signal lines: a TX line is used to output a signal from a device, and the RX line is used to input the signal. There is also a common ground line for both devices, Fig. 14.1 left. The timing diagram of the typical signal used to transfer character 'A' (ASCII: 65_{10} or 0x41) from device A to device B is given in Fig. 14.1, and would appear on the upper line TX -> RX between devices.

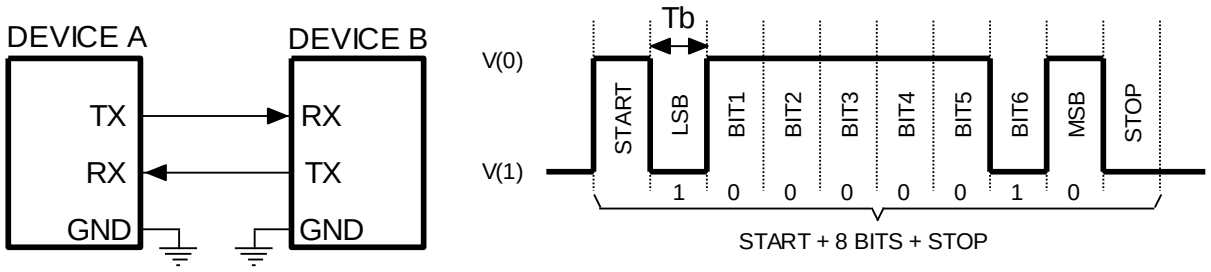


Figure 14.1: A serial communication conforming to RS232 protocol

The standard defines voltage levels V(0) to be at least +5V at the transmitting end of the line TX, and is allowed to degrade along the line to become at least +3V at the receiving and of the line. Similarly voltage level V(1) must be at least -5V at TX, and at least -3V at RX. The standard also defined the upper limit for these voltages to be up to ±15V. Logic high is transferred as V(0). The microcontroller cannot handle such voltage levels, so typically a voltage level translator is inserted between the microcontroller and the connector where the RS232 signals are available; the microcontroller implements so-called TTL version of RS232 standard.

The standard defines the number of bits to be transferred within one pack, Fig. 14.1 right, as eight for regular transmission, and nine for special purposes. The duration *Tb* of each bit defines the speed of transmission and is called the baud-rate. The typical baud-rate is 9600 bits per second (Baud, Bd), and the time *Tb* equals 104.16μs. Other baud rates are: 19200 Bd, 38400 Bd, 57600 Bd, and 115200

Bd. These are defined in standard, but used less frequently. The beginning of the pack of bits is signaled by a so called 'START bit', which has value 0 by definition. Its duration is equal to *Tb*. The pack of bits is terminated by so called 'STOP bit' with a typical duration of *Tb*, bat can also last either 0.5, 1.5 or 2 *Tb*, depending on the configuration. The complete transmission of a byte at typical baud rate of 9600 Bd takes 1.0416 ms.

To verify the validity of the transmission the protocol RS232 provides a so called "parity bit". A single bit is added by the transmitter before the stop bit, and its value is configured to produce either odd or even number of ones in a string. The number of ones in a string can be verified at the receiving end, and if it does not match the required value, the transmission should be repeated. Other, higher level protocols to ensure the valid transmission, can be implemented in software. The protocol RS232 also accounts for testing if the receiver is capable of receiving incoming bytes and defines two additional wires called RTS (Request To Send) and CTS (Clear To Send) between devices. We will not use any of these options in our experiments.

14.2. The hardware – serial communication

The microcontrollerSTM32F407VG includes up-to six hardware modules to deal with RS232 signals. Some of the modules additionally implement other communication protocols, like I2C, CAN, SPI; module named USART3 (Universal Synchronous Asynchronous Receiver Transmitter) will be used in this experiment. Its detailed description can be found in reference manual RM0090, chapter 30. The voltage level translator is added between the PC and the microcontroller since the BaseBoard accepts only TTL version of the RS232 signals TX and RX. The voltage translator can be either connected to the RS232 port of the personal computer or it can derive RS232 signals from the USB port of a personal computer. The signals TX and RX signals are available at connector K600, pins 5 and 4 respectively to ease the connection of the FT USB-RS232 3V3 interface. The RS232 signals RX and TX are available at microprocessor pins as alternate functions replacing the regular port pins, and those must be properly configured.

The block for handling the serial communication is complex, and we will not discuss its options based on the implementations in hardware here. Instead, we will use the CMSIS functions for the configuration of the hardware and will list the proper configuration procedure and comment the use of functions and their arguments.

14.3. The software – serial communication

The software presented demonstrates the use of serial communication. The main part of the program is listed below. The listing starts with the inclusion of CMSIS functions in lines 1 to6; note the include statement in the fourth line for the USART module. A pointer to a character string is declared next, the string will be used for communication purposes. The main function includes three calls to configuration functions for LED diodes on board, four pushbuttons and the LCD display; all these functions were already described in previous chapters and will not be discussed here again. The last configuration function in line 15 deals with the complete configuration of the USART module, and is new here. The execution of the program proceeds into the endless loop, where it periodically checks the state of pushbutton S370. If pressed, the green LED is turned on to verify the continuous execution of the loop.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_usart.c"                                                    // 4
#include "LCD2x16.c"
#include "dd.h"
```

```
char *OutString;        // string must be terminated by '\0'                                    // 8

void main(void) {

  LEDinit();                                                                                   // 12
  SWITCHinit();                                                                                // 13
  LCD_init();                                                                                  // 14
  USART3init(921600);                                                                          // 15

  while (1) {
    if (GPIOE->IDR & S370) LED_GR_ON; else LED_GR_OFF;                                         // 18
    for (int i = 0; i<100000; i++) {};          // waste time                                  // 19
  };
}
```

All communication tasks are processed within an interrupt function. The interrupt function is used for a good reason: the RS232 communication is rather slow, and it would be a complete waste of processor power to let it wait for signals from RS232. Let it rather do more important work (not in this example), and jump to interrupt function dealing with RS232 only when necessary.

As any other peripheral module in the microcontroller the USART also needs to be configured before used. There are actually three peripherals affected: port D is used to pass the TX and RX signals, the USART module is used to generate and receive RS232 signals, and the controller NVIC is used to process interrupt requests from the USART module. All three must be configured, corresponding function calls to CMSIS functions are put into a configuration function named “USART3init()”. The function requires one argument: the baud rate, and the complete listing of the function is given below.

```
void USART3init(int BaudRate)   {
GPIO_InitTypeDef        GPIO_InitStructure;                                                  // 2
USART_InitTypeDef       USART_InitStructure;                                                 // 3

  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);                                      // 5
  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_8 | GPIO_Pin_9;                                   // 6
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AF;                                              // 7
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;                                          // 8
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;                                          // 9
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;                                             // 10
  GPIO_Init(GPIOD, &GPIO_InitStructure);                                                     // 11
  GPIO_PinAFConfig(GPIOD, GPIO_PinSource8, GPIO_AF_USART3);                                  // 12
  GPIO_PinAFConfig(GPIOD, GPIO_PinSource9, GPIO_AF_USART3);                                  // 13

  RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART3, ENABLE);                                     // 15
  USART_InitStructure.USART_BaudRate            = BaudRate;                                  // 16
  USART_InitStructure.USART_WordLength          = USART_WordLength_8b;                       // 17
  USART_InitStructure.USART_StopBits            = USART_StopBits_1;                          // 18
  USART_InitStructure.USART_Parity              = USART_Parity_No;                           // 19
  USART_InitStructure.USART_Mode                = USART_Mode_Rx | USART_Mode_Tx;             // 20
  USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;            // 21
  USART_Init(USART3, &USART_InitStructure);                                                  // 22

  USART_ITConfig(USART3, USART_IT_RXNE, ENABLE); // enable IRQ on RX                        // 24
  NVIC_EnableIRQ(USART3_IRQn);   // Enable IRQ for USART3 in NVIC                            // 25

  USART_Cmd(USART3, ENABLE);                                                                 // 27
}
```

Two data structures for two call of CMSIS functions are required, these data structures are declared at the beginning of the function in lines 2 and 3.

The configuration starts with setting-up of the port D in lines 5 to 13 of the listing. The clock for port D is first enabled, then the data structure is initialized to use pins 8 and 9 (in line 6) to serve as alternate functions (in line 7); this data structure is used in line 11 to configure the pins. Next in lines 12 and 13 the correct alternate function is selected by two calls to functions "GPIO_PinConfig()", where the USART3 function is selected.

The configuration continues with setting-up of serial module USART3, lines 15 to 22. The operation of this module is again defined by the content of several configuration registers within the module, and to avoid problems in identifying correct bits in these registers and their function we use the CMSIS function "USART_Init()" instead (line 22). The function is listed in the source file "stm32f4xx_usart.c", and requires the use of a data structure 'USART_InitStructure', declared in the header file "stm32f4xx_usart.h", lines 54 to 81. Brief instruction on the use of USART module are given as usually at the beginning of the source file.

The clock for the USART module is enabled first in line 15. Note that the bit responsible for enabling the clock is not located in the same register as the bit responsible for the clock of ports since the USART module is connected to a different bus within the microcontroller, and the name of the function called differs. The details can be found in reference manual RM0090.

Lines 16 to 21 are used to initialize members of the data structure required in function call in line 22. They are initialized as follows.

- The first member '.USART_BaudRate' has a self-explanatory name. It can be any number, and the CMSIS function will do its best to configure frequency dividers inside the USART module to achieve the baud rate desired. The formula for the calculation is given in the header file, line 56, and the detailed description is to be looked for in the reference manual RM0090. In this example the value is passed into the initialization function as a parameter.
- The second member '.USART_WordLength' defines the number of bits per package sent, and can be either 8 or 9, see the header file, lines 128 and 129 for definitions.
- The third member '.USART_StopBits' defines the duration of the STOP bit, and can be one of the four values listed in the header file, line 141 to 145. Here we select the standard one STOP bit.
- The next member '.Parity' defines the use of the parity bit as one of the options listed in the header file, lines 157 to 159. In this example we do not implement parity checking, so 'USART_Parity_No' is used to initialize this member.
- The fifth member '.USART_Mode' enables the use of the module to receive and/or transmit. In our case both directions are enabled.
- The last member '.USAT_HardwareFlowControl' enables the use of additional signals CTS and RTS, as defined in the header file, lines 181 to 184. We do not use these additional lines, so the keyword 'USART_HardwareFlowControl_None' is used to initialize the member.

The pointer to this data structure is used in line 22 as an argument to a function call. Note that the call configures module USART3, as specified by the first argument.

We have decided to use interrupts to deal with the needs of the USART3 module. The first reason for the interrupt request is obvious: when a module receives a character the processor should remove it from the module and do something with it. In any case there may be more characters coming, and the next one could overwrite the one that was just received, so processor better move it from the USART module fast! On the other hand we do not want to waste valuable processor time by writing a

program that makes processor wait for commands from the USART module or to make processor periodically poll the module for commands.

We must therefore activate the interrupt processing for the incoming characters. This is done in two steps in lines 24 and 25. First the USART module is allowed to issue interrupt request signals, and these interrupt requests should appear due to the reception of a character; the interrupt processing for the RX function is enabled at the module USART3 in line 24. Next the interrupt controller block NVIC is instructed to react on interrupt requests from the USART3 module in line 25. Obviously, there must be an interrupt function prepared in order for interrupt processing to work.

The last line of the configuration function simply enables the USART3 module.

A response to the interrupt request requires an interrupt function. An example of such function to handle interrupts from USART3 is presented next.

```
void USART3_IRQHandler(void)     {

  // RX IRQ part
  if (USART3->SR & USART_SR_RXNE) {   // if RXNE flag in SR is on then                          // 4
    int RXch = USART3->DR;            // save received character & clear flag                  // 5
    LCD_uInt16(RXch, 0x05, 1);        // show ASCII on LCD                                     // 6
    if (RXch == 'a') LED_BL_ON;                                                              // 7
    if (RXch == 'b') LED_BL_OFF;                                                             // 8
    if (RXch >= 'A' && RXch <= 'Z') USART3->DR = ++RXch;  // echo this character             // 9
    if (RXch == 'c') {                // if 'c' => return string                             // 10
      OutString = "ABCDEFGH";         // Init string & ptr                                   // 11
      USART_ITConfig(USART3, USART_IT_TXE, ENABLE);                                          // 12
      USART3->DR = *OutString++;      // Send first character                                // 13
    };
  };

  // TX IRQ part
  if (USART3->SR & USART_SR_TXE) {    // If TXE flag in SR is on then                        // 18
    if (*OutString != '\0')           // if not end of string                                // 19
      USART3->DR = *OutString++;      // send next character & increment pointer             // 20
    else                              // else                                                // 21
      USART_ITConfig(USART3, USART_IT_TXE, DISABLE);                                         // 22
  };

}
```

The interrupt function is named as required in the interrupt vector table (UART3_IRQHandler), and neither receives not returns any variables. Any variables used and expected to last longer than the execution of the interrupt function must be declared as global. The function is intended to retrieve the received characters from the USART3 module; this is the ‘RX IRQ part’. We are also going to send a string of bytes to demonstrate the capabilities of the microcontroller; such string, actually a pointer to a string, was already declared in the first listing (char *OutString). Such string is expected to be terminated by a ‘\0’, a standard C-language termination character, and the corresponding software is listed in ‘TX IRQ part’.

There are many possible reasons to interrupt the execution of a regular program. When a new byte is received by the UART3, it appears in the USART Data Register (USART3->DR), and should be removed from there and handled before it gets overwritten by a next byte received. This is so-called receive interrupt. There are other reasons for UART3 to request attention and issue an interrupt request; some of them will be dealt with later, they are all listed in the source file, lines 254 to 256 and described in the reference manual RM0090. However, there is only one interrupt request signal available to be

utilized by USART3, and the reason for the interrupt request must be known in the interrupt function. The code of the reason is stored in status register of the USART3. Bit called 'RXNE' (bit 5, Receiver data register Not Empty) is set when the receipt of a byte causes the interrupt request, and this can be checked within the interrupt function. The body of the interrupt function therefore starts by testing this bit in line 4; if it is set then the reason for the interrupt is a byte waiting in the USART3 data register, and this byte must be read from there. This gets done in the 5th line of the listing. This read from the data register simultaneously clears the bit RXNE, and the USART3 is ready to receive next byte.

Once the received byte is safely retrieved the interrupt function compares it with some predefined values in lines 7 to 9. If the value is equal to ASCII 'a' then the blue LED is turned on. If its value is equal to ASCII 'b' then the same LED is turned off. If the value of the byte received is between ASCII 'A' and ASCII 'Z' then the byte is echoed back to the sender by a simple write back into the data register of the same USART.

A user program might require a string of bytes is to be transferred. This possibility is demonstrated next. Since many bytes are to be transferred, this is expected to take some time. Unfortunately, the data register in the USART module must be written byte by byte only after the previous byte gets successfully sent over the TX line out of the microcontroller. The processor time would be wasted if the processor is programmed to periodically check whether the module USART is ready to process the next byte, so an interrupt function should be used for transmission as well.

The procedure is as follows. Initially the string of bytes to be transferred is prepared. The string should be terminated with a unique byte as in this example (the C-language automatically terminates a string by '\0' character) to ease the end-of-string detection or the length of the string should be known in advance. Another interrupt request should be enabled, this interrupt request shall be issued when a byte is fully transmitted and the next one is allowed to be written into the data register of the USART; this is done by enabling 'USART_IT_TXE' option, CMSIS function "USART_ITConfig()". The last thing is to write the first byte of the string into the data register USART3->DR sending it out of the microcontroller, increment the pointer to the string of bytes and exit the interrupt function. These three steps are implemented as the fourth option, lines 10 to 14, when a byte ASCII 'c' is received.

Wwhen the first byte of the string is transferred over the USART3, an interrupt request calls the interrupt function again, but this time bit TXE (Transmit data register Empty) is set signaling next byte can be written into the data register, and the bit RXNE is cleared. The second IF statement is implemented in listing, line 18, to check the status of bit TXE. When set, the current element of the string is checked against the termination value ('\0'), line 19.

- If the current element of the string differs from the termination value it gets written into the data register, therefore sent through USART3 out of the microcontroller, and the pointer to the string of bytes is incremented. The interrupt request for transmission is left enabled, so the sending of characters can repeat.
- If the current element is equal to the termination value then the interrupt requests for transmission are disabled by a call to the function "USART_ITConfig".

The operation of the program can be verified by the use of HyperTerminal (Windows, pre-7 edition), or one of the freely available terminal emulators.

Some caution should be exercised when such program is debugged. Placing a breakpoint into the interrupt function stops the execution of the program and simultaneously causes a read from registers within the microcontroller by the debugging software. The read includes the data registers of the USART. Some flags may be affected by this read, for instance the flag on TXE and RXNE, causing wrong execution of the interrupt function in step-by-step mode of the debugger! Such traps are difficult to

identify and the author of this text spent one afternoon in looking for an error in this program, but there was no error at all! Errors were introduced by misplaced breakpoints, and the program worked fine after the breakpoints were moved to a better place.

Please note that the BaseBoard and the STM32F4-Discovery board accept the TTL version of the RS232 signals. These can be obtained for instance by the use of a USB to TTL-RS232 converter cable, like FT USB-RS232 3V3. Note that the converter must be designed for 3.3V, and that the version for 5V might not work correctly!

Signals available at the RS232 D-9 connector at the back of a personal computer can damage the board! Do not use them for this experiment!

---

15. Serial communication – sampling by ADC

Techniques presented in previous chapters will be combined here to connect the microcontroller to a personal computer using the RS232. The program for the microcontroller will initialize USART3 and then wait for a command coming through RS232 connection from the PC. When a suitable command (character 'a') is received the ADC and timer TIM5 will be enabled to start a periodic sampling of two analog input signals. The time interval will be defined by the reload value of the timer, and the predefined number of samples will be stored into an array. The content of the array will be sent through UART to the personal computer when a suitable command (character 'b') is received.

15.1. The software

The hardware used in this example and its configuration was discussed in previous chapters, and will not be repeated here. This chapter describes only the additions to programming and possible modifications to give a usable demo. The chapter does give the new interrupt functions to achieve the goal, and reports the quality of the ADC used.

The listing of the main part of the program is given below. Initially all CMSIS files needed by the compiler are included, then an array named 'Results[]' for storing the results from ADC is declared as global in line 9; it will be used in interrupt function. The array consists of characters and this might look strange. However, characters can be sent using the RS232. The results from ADC are 12 bits in width, so a function for storing results has to split the 12 bits into two sections and store both in consecutive characters of the array. The length of the array is 65536 elements, therefore 32k results from an ADC can be stored. A pointer 'ResultsPointer' to the array of results is needed and is declared and initialized in line 10, also as global.

The main part of the program calls four functions to configure the hardware. These functions are exactly the same as described in previous chapters. Note that a very fast baud rate was selected. This is the fastest standard baud rate on a PC, and can be used only if the wire between the PC and the microcontroller is relatively short, about 1 m is safe to use. Note also that the time interval between two consecutive interrupt requests from the timer is fixed to 100μs, and hardware triggering of the start of conversion is used.

All processing is done in either in hardware or interrupt functions, so the processor enters an empty endless loop after the configuration. Everything is set-up to start the periodic sampling but the timer is not enabled and there is no sampling yet, only the USART3 is ready to receive a byte through RS232. The processor might as well be put into sleep mode conserving power; this is not implemented here.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_adc.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_usart.c"
#include "stm32f4xx_tim.c"
```

```
#include "dd.h"

char Results[65536];                                                    // 9
int ResultsPointer = 0;                                                 // 10

void main(void) {
  LEDinit();                                                            // 13
  USART3_init(921600);                                                  // 14
  ADCinit_T5_CC1_IRQ();                                                 // 15
  TIM5init_TimeBase_CC1(840);           // 840 == 100us                 // 16

  while (1) {                           // endless loop
  };
}
```

The processor keeps on executing an endless empty loop until a character is received by the USART. The essence of the program in hidden in interrupt functions.

When a correct byte is received by the USART3, a sampling should start. The programmer should prepare an interrupt function to be called on an interrupt request from the USART3 as shown already in chapter 14. This function is again composed of two parts, the upper one for handling the receipt of a command character, and the lower one for handling the sending. The complete listing is given next.

```
void USART3_IRQHandler(void)     {

  // RX IRQ part
  if (USART3->SR & USART_SR_RXNE) {  // if RXNE flag in SR is on then
    int RXch = USART3->DR;       // save received character & clear flag
    if (RXch == 'a') {                                                  // 6
      LED_BL_ON;                // blue ON                              // 7
      ResultsPointer = 0;       // init pointer to results              // 8
      TIM_Cmd(TIM5, ENABLE);                                            // 9
    };
    if (RXch == 'b') {                                                  // 10
      USART_ITConfig(USART3, USART_IT_TXE, ENABLE);                     // 11
      ResultsPointer = 0;                                               // 12
      LED_RD_ON;                // red ON                               // 13
      USART3->DR = Results[ResultsPointer++];   // Send first character  // 14
    };
  };

  // TX IRQ part
  if (USART3->SR & USART_SR_TXE) { // If TXE flag in SR is on then       // 19
    if (ResultsPointer < 65536)    // if not end of string               // 20
      USART3->DR = Results[ResultsPointer++]; // send next byte & pointer++ // 21
    else {                         // else
      USART_ITConfig(USART3, USART_IT_TXE, DISABLE);                    // 23
      LED_RD_OFF;                  // red OFF                            // 24
    };
  };

}
```

If the received character equals ‘a’ then the sampling should start. First the blue LED is turned on to signal the start of sampling in line 7, then the pointer into the array is initialized to point to the zeroth element in line 8, since this is the beginning of the place to store results of sampling. Next the timer TIM5 is enabled, and the execution of this interrupt function is terminated; the processor returns to

the dull part: the execution of the endless empty loop. Any further action depends on commands received by the USAT3 or interrupt request from the ADC.

When the content of timer TIM5 becomes equal to 1 its output OC1 triggers the start conversion at both ADCs, see chapter 14. Once results are available the ADC issues its own interrupt request, and this causes the execution of the interrupt function reserved for ADC (this function is named "ADC_IRQHandler"), as given bellow.

```
void ADC_IRQHandler(void)       {
  int Ra = ADC1->DR;                                                    // 2
  int Rb = ADC2->DR;                                                    // 3
  if (ResultsPointer < 65535)    {                                      // 4
    Results[ResultsPointer++] = (Rb & 0x0f00) >> 8;                     // 5
    Results[ResultsPointer++] = (Rb & 0x00ff);                          // 6
  }
  else  {
    LED_BL_OFF;                 // blue LED OFF                         // 9
    TIM_Cmd(TIM5, DISABLE);                                             // 10
  };
}
```

The results from both ADCs are now available, and they are latched into two temporary variables "Ra" and "Rb", lines 2 and 3; the latching also clears the interrupt request flag in ADC module to prevent immediate re-entrance into the same interrupt function. One of the results ('Rb') is split into two bytes and stored into two consecutive elements of the array in lines 5 and 6. At the end of this saving the pointer 'ResultsPointer' points to element number 2 of the array.

A predefined number of samples is to be taken and stored, in our case 32768 (after taking 32768th sample the pointer points to element 65536). If pointer is less than 65535 then the sampling is to continue, and no further actions are needed; this is checked in line 4. The execution of the interrupt function can be terminated, and the processor returns to execute the endless loop until next interrupt request when the action described two paragraphs above is repeated.

However, if the all the required samples were already taken the pointer equals 65536; the sampling should stop. In this case lines 9 and 10 get executed. Here the blue LED is turned off to signal the end of acquisition, and the timer TIM5 is disabled preventing any further start conversion pulses for ADCs. This terminates the sampling phase, and the processor is now waiting for the interrupt requests only from USART3.

When next character is received by the USART3, the interrupt function for the USART3 is entered again. If the character received equals 'b', then lines 11 to 14 are executed. These lines start the process of sending results from the array of results through the RS232. The USART3 interrupt request line for the end of transmission is enabled first, then the pointer to the array is initialized to 0. The red LED is turned on in line 13 to signal the start of transmission, and the transmission is commenced in line 14 by sending the first element of the array to the USART3 data register, and the pointer to array gets incremented. This finishes the execution of the interrupt function, and the processor returns to the empty loop.

Once the USART3 is ready to accept the next byte of results it issues an interrupt request again, and the processor re-enters the interrupt function for USART3. The checking of the cause of interrupt in line 19 confirms that the next byte is to be sent. This confirmation is followed by a checking if all elements of the array have been sent in line 20. If pointer is less than 65536 this is not true and the next element is passed to the USART3, and the processor returns to the empty loop. Once all elements are sent the checking in line 20 diverts the execution to line 23, where further interrupt requests for sending are disabled and the red LED is turned off.

This program was used to test the quality of the ADC, and the results are presented in Fig. 15.1 to 15.3. The diagram in Fig. 15.1 shows results obtained when a slowly decreasing signal at about ¼ of the full scale was applied to the input of the ADC. The version presented is idealized to an extent, this is explained in the following paragraph. One can observe the resolution of the ADC: the spread of the results is small, and equals to about ±1.5 LSB. We are therefore dealing with about 10.5 - bit ADC, crudely speaking.

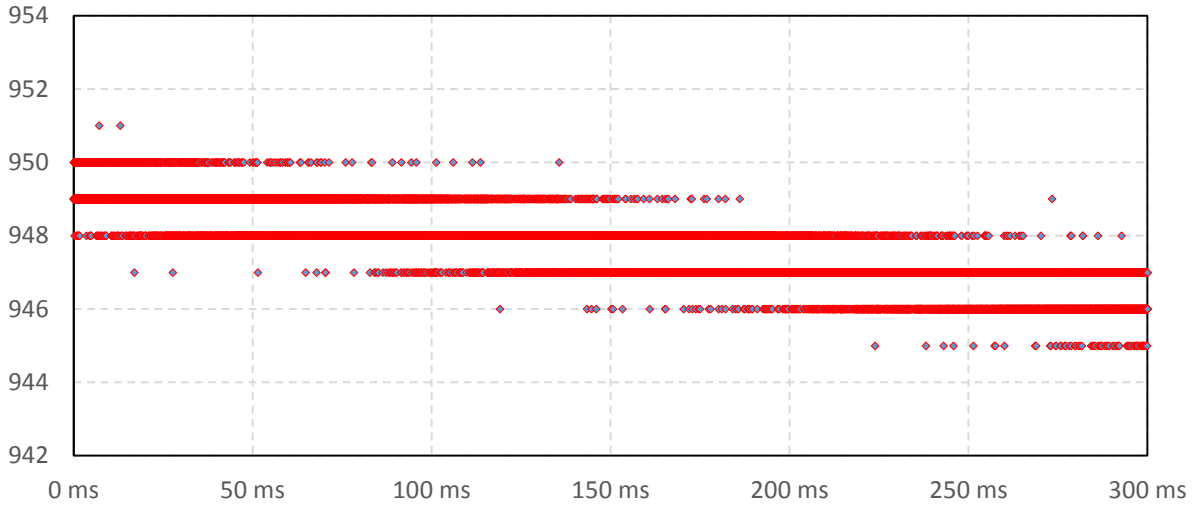


Figure 15.1: A slowly decreasing input signal as sampled by the board and this software; the spread of results is small, all fall within four (sometimes even only three) adjacent values.
Horizontal axis – time, vertical axis – result of measurement as coming from the ADC (0 to 4093)

The next diagram in Fig. 15.2 gives the distribution of samples when a constant voltage of about ¼ of the full scale voltage is applied; again, the spread of the results is small. About 80% of all results (30000 samples were accounted for) have the same value of 971. About 6% of results have value of 970, and about 14% are 972. However, for some reason results far out of these values were also obtained. They are not numerous and account for about 0.16%, and can be best seen on Fig. 15.3, the actual result of sampling, no idealization. The spikes are ascribed to the Discovery board with relatively poor analog layout, but may have other reasons (to be investigated).

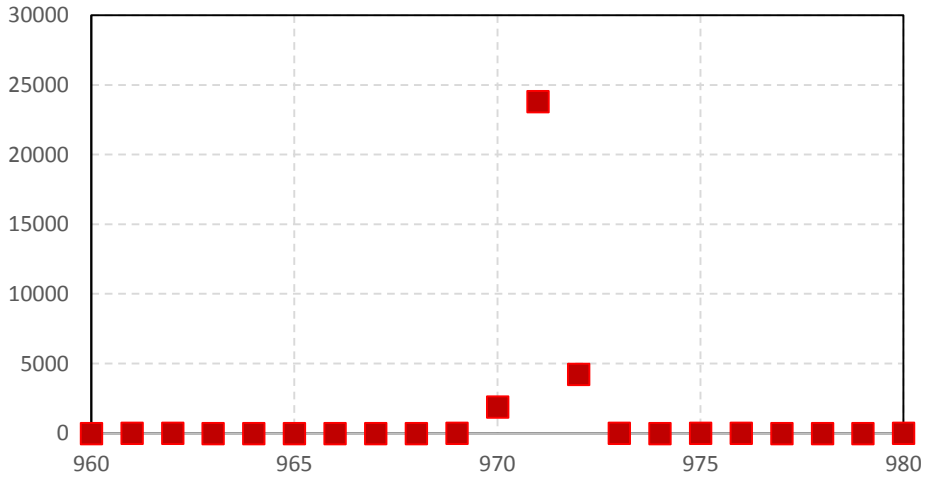


Figure 15.2: The distribution of samples, constant input signal; horizontal axis – measurement result, vertical axes – frequency of occurrence.

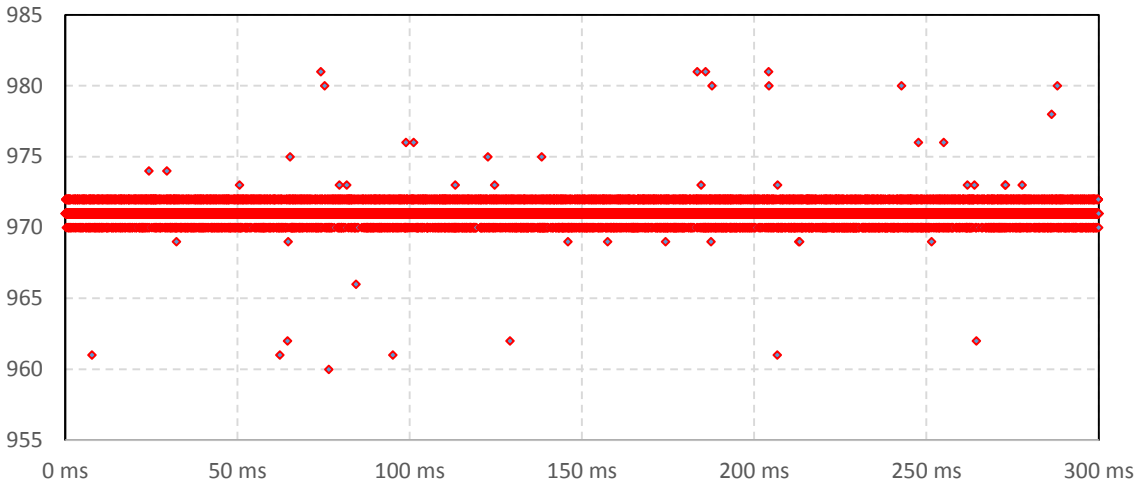


Figure 15.3: A constant signal is sampled, most of the results fall within three channels, but some of them are very far from the expected value. Horizontal axis – time, vertical axis – result.

Next a sinusoidal signal was applied to the input to the board; the signal had a frequency of 5 kHz and amplitude of about 1.5V, biased at about ½ of the ADC range. The last diagram in Fig. 15.4 gives a normalized frequency spectrum of this signal. The spectrum was obtained by commercial software for data processing and presentation (DaDisp & Microsoft EXCEL). Calculation involved the use of Kaiser window function and standard algorithm for the FFT (Fast Fourier Transform). It can be seen that the first harmonic is about 60 dB below the fundamental, and higher harmonics are even lower. It seems a bit strange to have the even multiples harmonics so strong, and this shall be investigated still.

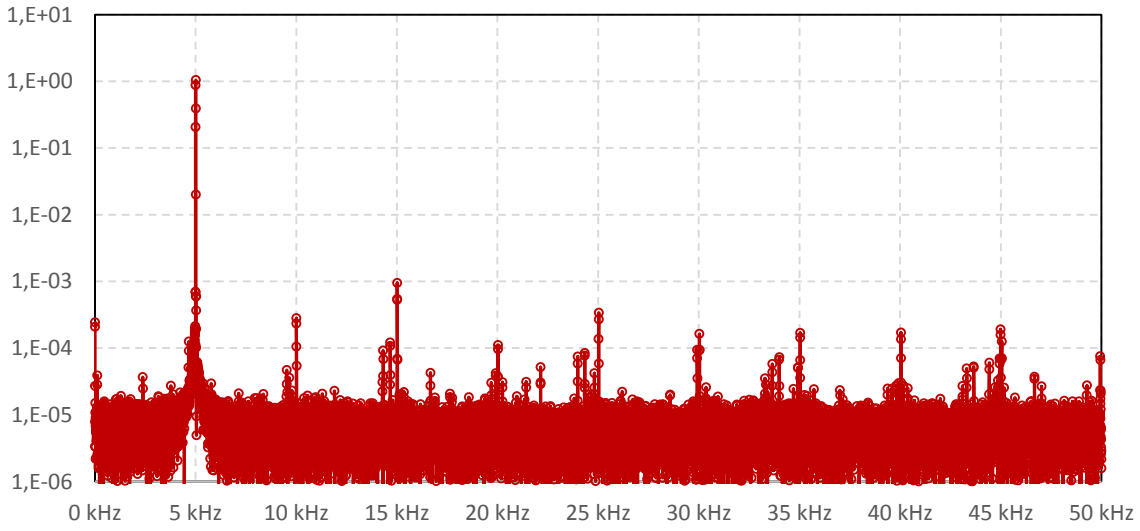


Figure 15.4: The spectrum of the sine wave signal as obtained by the board; horizontal axis – frequency, vertical axis – normalized.

16. Signal generation using DDS

A technique known as Direct Digital Synthesis (DDS) will be implemented to demonstrate the use of interrupts and DAC to generate a signal with the desired frequency.

16.1. DDS technique

Consider a 16-bit wide register. Its content is increased periodically (the period is given as T_P) by a factor *K*. Obviously the time *T* needed to overflow the register equals:

$$T = T_P \cdot \frac{2^{16}}{K}$$

If we keep increasing the content of the register then the overflows will repeat at regular time intervals, and the frequency of overflows *f* can be calculated as:

$$f = \frac{1}{T} = \frac{1}{T_P} \cdot \frac{K}{2^{16}} = f_P \cdot \frac{K}{65536}$$

where f_P represents the number of increases per unit of time. If we consider the content of the register as the output signal, and the factor *K* is small, then we get a sawtooth generator; its frequency is directly proportional to factor *K*. However, we can use the content of the register as a pointer to a table, which can be filled by any waveshape during the initialization process, like one period of a sinewave. If we consider the entry from a table pointed to by the content of the register as the output signal we have a sinewave generator. The frequency of the sinewave signal can be adjusted by the same small steps, and the amplitude can be adjusted by a multiplication of the retrieved entry by a constant before it gets passed to the DAC.

It does not seem rational to prepare a huge table with 65536 entries, since the neighboring entries would be quite close in value for a slowly varying signal like sinewave. It is more rational to prepare a smaller table, and use only the bits from the upper part of the register as a pointer. The decision on the table length depends on the required precision, and we arbitrary decide to use a table with 4096 entries in this example. It would be possible to reduce the size of this table to ¼ by exploiting the symmetry of a sinewave, but we will not do this for the reason of simplicity. The block diagram of the

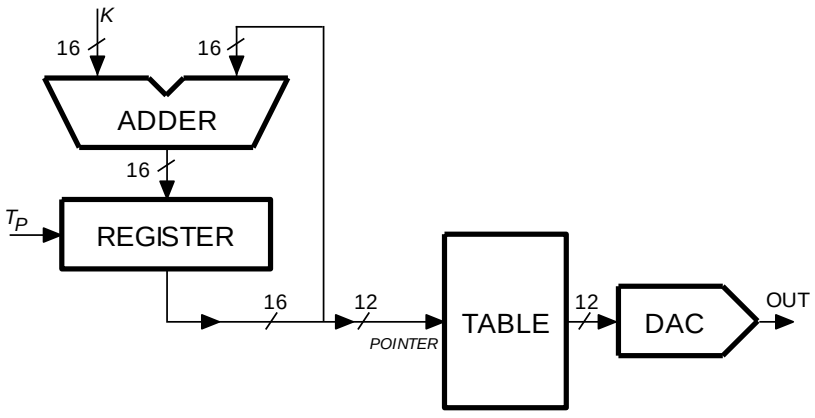


Figure 16.1: A hardware implementation of the DDS technique

hardware that could be used as a DDS generator is shown in Fig. 16.1.

The same technique is widely used in contemporary digital receiver units, and width of registers is increased to about 40 bits as is the frequency of increasing to some hundreds of MHz to achieve a stable output signal with a frequency of above 100 MHz that can be adjusted by few mHz! By expanding this technique it is very simple to generate a modulated signal (AM, FM, PM, …), and also to define new shapes of signals as well as to generate multiple signals with different frequencies. There are specialized integrated circuits available to do just this.

16.2. The software implementation

Such structure is easy to implement in a microcontroller. One only needs a timer and associated interrupt function. The timer defines the period T_P, and periodically calls an interrupt function, where a variable declared as an unsigned short integer (16 bits) gets increased by a factor *K*.

If we select the frequency of interrupt requests from the timer (f_P) at 100 kHz and a register with 16 bits, then the frequency of overflows is defined by:

$$f = 1.525879\ \mathrm{K}$$

Changing factor *K* by one changes the frequency of the generated signal for about 1.5Hz. If we intend to generate audio signals with frequencies from 20 Hz to 20 kHz then the factor K must vary from 13 to 13107. The listing of the main part of the program is given below.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_dac.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_tim.c"
#include "dd.h"
#include "LCD2x16.c"
#include "math.h"

int Table[4096], ptrTable, Am = 255, k = 655;                                   // 10

int main ()  {
int sw, Fp, j;
  for (ptrTable = 0; ptrTable <= 4095; ptrTable++)                              // 14
    Table[ptrTable] = (int)(1850.0 * sin((float)ptrTable / 2048.0 * 3.14159265));  // 15

  SWITCHinit ();                                                                // 17
  LCD_init (); LCD_string("Frq=", 0x01); LCD_string("Hz", 0x0d);                // 18
  LCD_string("Amp=", 0x41);                                                     // 19
  DACinit ();                                                                   // 20
  TIM5init_TimeBase_ReloadIRQ(840);           // 840 == 10us                    // 21

  while (1) {                                 // endless loop                   // 23
    sw = GPIOE->IDR;                                                            // 24
    if ((sw & S372) && (Am <   255)) Am++;                                      // 25
    if ((sw & S373) && (Am >     1)) Am--;                                      // 26
    if ((sw & S370) && (k  < 13107)) k++;                                       // 27
    if ((sw & S371) && (k  >    13)) k--;                                       // 28
    Fp = (int)(1.0e5 * (float)k / 65536.9);                                     // 29
    LCD_uInt16(Fp, 0x08, 1); LCD_uInt16(Am, 0x48, 1);                           // 30
    for (j = 0; j<200000; j++){};      // waste some time                       // 31
  };
}
```

The listing starts with the inclusion of necessary files. Note the file “math.c” needed for calculation of the sinusoidal wave shape. The declaration of the table ‘Table[]’ with 4096 integer elements, the pointer ‘ptrTable’ to this table, and some variables to define the amplitude *Am* and factor *K* follows in line 10. All of these must be declared as global since they are used in the interrupt function. The processor then fills the table with entries of one period of a sine wave, lines 14 and 15. The values range from -1850 to +1850, allowing for some headroom to accommodate the saturation voltage of the operational amplifier at the output from DACs. Floating point arithmetic must be used to calculate ‘sin’ function. Next is the configuration of all peripherals used: ports, DAC, timer, and LCD. The functions are taken from previous chapters and will not be discussed here again; note only that the time interval between successive interrupt request for timer TIM5 is set to 10μs (840).

The processor continues with the execution of the endless ‘while’ loop. In order to make program more flexible (and motivating) we shall make the amplitude and frequency of the generated signal changeable by means of four pushbuttons. Two can be used to increase/decrease the amplitude, and two to increase/decrease the frequency.

The processor first reads pushbuttons S370 to S373 in line 24, and then decides upon changing of values in four consecutive lines from 25 to 28. In line 25, for instance, the processor checks the state of pushbutton S372; if it is pressed and the current value of multiplier for the amplitude of the output signal is less than 255, the value of this multiplier is increased. The next line checks the pushbutton S373 and the current value of multiplication factor. If conditions are favorable, it decreases the value of multiplication factor. Similar operations are performed in lines 27 and 28 for the frequency.

Line 29 is used to calculate the frequency of the generated signal. The division factor seems strange as it should read 65536. However, this value was used due to the calibration of the readout. The commands in line 30 are used to write the calculated frequency and amplitude to the LCD screen.

The DDS technique is implemented in the interrupt function for timer TIM5, given below.

```
void TIM5_IRQHandler(void)       {
  TIM_ClearITPendingBit(TIM5, TIM_IT_Update); // clear interrupt flag                    // 2
  ptrTable = (ptrTable + k) & 0xffff;                                                   // 3
  DAC->DHR12R1 = (Am * Table[  ptrTable >> 4               ]) / 256 + 2048;             // 4
  DAC->DHR12R2 = (Am * Table[((ptrTable >> 4) + 1024) & 4095]) / 256 + 2048;            // 5
}
```

It is properly named as required by the interrupt vector table as “TIM2_IRQHandler”. The body starts by clearing the interrupt flag in timer TIM5 (line 2) and proceeds to calculating the new pointer value. A factor *K* is added to the pointer, and its value is bound to 16 bits by AND-ing it with value 65535. The pointer is used in the next line to retrieve correct element from the table, and only upper 12 bits of the pointer are used (this is the “ >> 4 ” operation). The retrieved element is multiplied by a variable Am (amplitude), and the resulting number divided by 256 to remain in the range of the DAC. The DAC can handle positive numbers only, so half of the DAC range (2048) is still added before sending the number to its destination.

In order to show the possibility of phase modulation (and also to allow further experiments with quadrature signals) the second DAC is used to generate a sine wave signal which is 90 degrees out of phase with the already generated signal. This is accomplished by retrieving the element from the table which is for ¼ of the table length away from the current pointer. Upper 12 bits of the pointer are taken, and 1024 is added to it. The sum might point outside of the table, so the sum is corrected to remain within the bounds of the table by AND-ing it with the pointer to the last valid element (4095), the rest is the same as for the first DAC. The complete interrupt function takes about 500ns to execute, much less than the available 10μs. The frequency of interrupts could be increased to allow the generation of smoother signals.

17. Serial communication - I²C

Two typical serial busses will be discussed and their use will be demonstrated in this and next chapter, these are I²C (Inter Integrated Circuit, IIC or I2C) and SPI (Serial Peripheral Bus). Only a simple variant of busses without the error checking will be implemented. The experiment requires a sensor that can communicate using the bus, and suitable accelerometers / gyroscopes will be used in our laboratory.

17.1. The I2C protocol - hardware

Sometimes sensors are distant to the microcontroller. In such case it might be unreasonable to send analog signal from sensor to a distant microcontroller containing an ADC due to the possible degradation of analog signal along the line. In would be better to include an ADC into the sensor, and pass the digital result of the conversion to the microcontroller. Digital signals are far less prone to degradation, and ADCs are simple and cheap to include into sensors. Such sensors are quite frequent at the present state of technology.

Digital signals coming from an ADC are 8 or 16 bits wide, and ADC chip alone needs additional control and status signals. When one considers the number of required wires and pins at the microcontroller to accommodate all these signals, one recognizes the need for a more efficient transfer of data into a microcontroller. Serial busses were invented to avoid these problems. They use considerably less wires to transfer the same information, and several devices can be connected to the same bus. The information is conveyed serially bit by bit in accurately defined time slots. To avoid problems with time slots and synchronization, a common clock signal is used by all devices connected to the same serial bus. The price one has to pay to use fewer wires is the prolonged time to transfer the same information and a rather complex set of rules to be obeyed when transferring information or devices connected to the common bus might rebel and refuse to transfer.

The I²C is the name of the bus that requires three wires between devices: a common ground and two wires for clock signal (SCL, Serial CLock) and data signal (SDA, Serial DAta). Several units may be connected to the same bus as shown in Fig. 17.1, but only one of the devices controls the dataflow; this device is called a master, others are called slaves. All devices have a so called “open collector” (“open drain”) outputs meaning that they can only pull a signal low, but never force it to go high. This is the task of two pull-up resistors, one for each wire to pull the line to logic high. Therefore, if no device requests the line to be low, the line remains high due to the pull-up resistor. If any or many devices turn on their outputs pulling the line low, the line becomes logic low. The value of a pull-up resistor depends on the

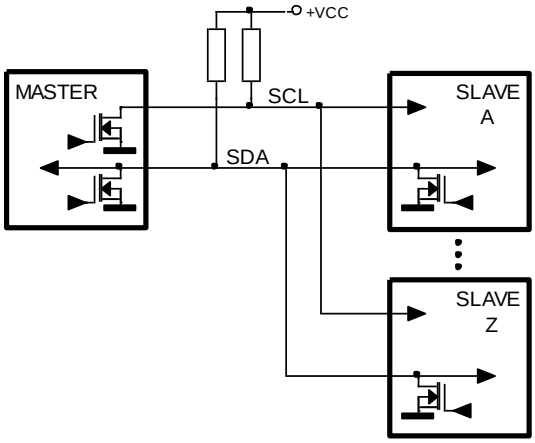


Figure 17.1: The I2C bus can be used to transfer data between the MASTER and one of the SLAVE devices

required speed, and is typically few kΩ. The logic levels are defined by the power supply +VCC of the devices connected to the bus and their technology; in our case the power supply is close to 3.3V.

The procedure to transfer the data over I^2C bus is described in I^2C protocol. There are four typical combinations of signals SCL and SDA:

a) Start combination. Both signals SDA and SCL are initially high. First signal SDA goes low, next signal SCL goes low, as in Fig. 17.2a, left. The important part is that signal SDA changes from high to low when signal SCL is high. The start combination defines the beginning of the transmission over the bus.

b) Stop combination. Both signals SDA and SCL are initially low. First SCL goes high, and then signal SDA goes high, as in Fig. 17.2b, right. The important part is that the signal SDA changes from low to high when signal SCL is high. The stop combination defines the end of transmission over the bus.

c) Bit transfer combination. Signal SDA (serial data) can be either low or high, and the signal SCL changes from low to high and then to low again forming one clock pulse on the SCL line. This combination clocks one data bit into the destination device, Fig. 17.2c. A string of eight (ten in special cases) bits is used to send the complete byte. The signal SDA must not change during the time signal SCL in high.

d) Acknowledge combination. The destination device is expected to confirm safe receipt of a byte. A string of eight data bits is followed by an acknowledge combination, where the originating device releases the SDA line, and the master device issues one clock pulse at the SCL line. The receiving device is supposed to pull the SDA line low during the clock pulse if it managed to receive eight bits successfully, Fig. 17.2d.

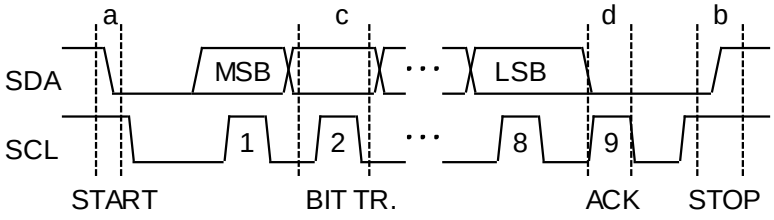


Figure 17.2: The signals on the I2C bus

There can be more than one slave device on the bus, and the master device can communicate with one slave device at a time. The protocol defines the addressing procedure to activate one of the slave devices. The I^2C standard allows the use of either 7-bit addressing or 10-bit addressing; we will demonstrate the use of 7-bit version.

Consider the situation where the master is writing into the slave, Fig. 17.3, top. The master first sends a start combination, followed by eight bits; here master defines consecutive values at the SDA line and issues eight clock pulses on the SCL line. Out of these eight bits first seven represent the address of the slave, and the 8^{th} bit is low signaling the writing into the slave. If a slave with the address specified is connected to the bus, then the slave confirms its presence by the acknowledge combination; the master sends the 9^{th} clock pulse and the slave pulls the SDA line low during the time of this pulse. From now on the master can send as many bytes to the selected slave as desired. Each byte is followed by the acknowledge combination, where the slave pulls the SDA line low telling the master that it is still able to receive bytes. When the transmission is complete, the master issues the "stop" combination releasing the slave. The address of the slave in this example is 0x37, and the data written is 0x57.

Similarly the reading from the slave into the master is started by a “start” combination and the addressing byte, Fig. 17.3, bottom; out of this first seven bits represent the address, and the 8th bit is high signaling the reading from the slave. If a slave with the address specified is connected to the bus, then the slave confirms its presence by the acknowledge combination; the master sends the 9th clock pulse and the slave pulls the SDA line low during the time of the pulse. From now on the master can read as many bytes from the slave as desired by issuing a pack of eight clock pulses for every byte read. Each byte is followed by the acknowledge combination, where the master pulls the SDA line low telling the slave that it has received the byte and that the slave is expected to continue sending bytes. The transmission is terminated by the master not acknowledging the receipt of the last byte and issuing the “stop” combination. The address of the slave is again 0x37, and it returns 0x39.

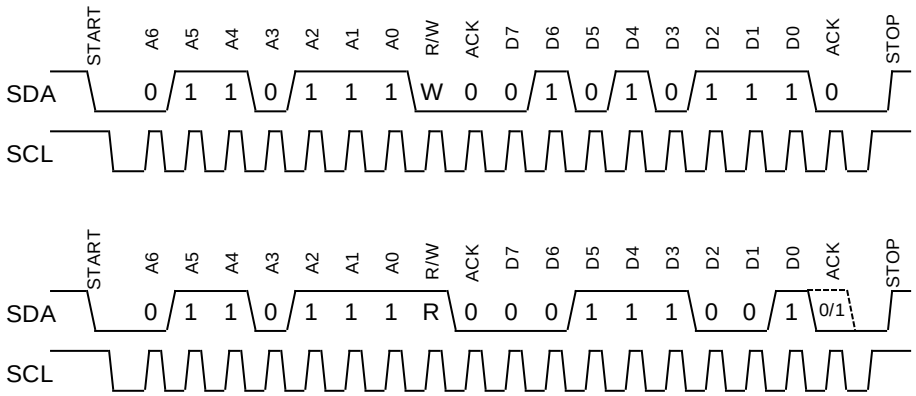


Figure 17.3: I2C bus signals during the writing into the slave (top), and reading from the slave (bottom)

Three blocks named I2C1 to I2C3 intended for I²C communication are built into the microcontroller. These blocks are identical and a simplified diagram for one of them is given in Fig. 17.4. Two pins at the microcontroller are required for lines SDA and SCL, and certain pins can be configured for alternate functions SDA and SCL. The I²C block includes two data related registers:

- the ‘Data Register DR’; the processor writes data to or reads data from this register, the register serves as an buffer between the processor and the actual hardware of the I²C block, and
- the ‘Data Shift Register DSR’; the data is transmitted serially, and this register is used to shift the data byte bit-by-bit out-of or in-to the block.

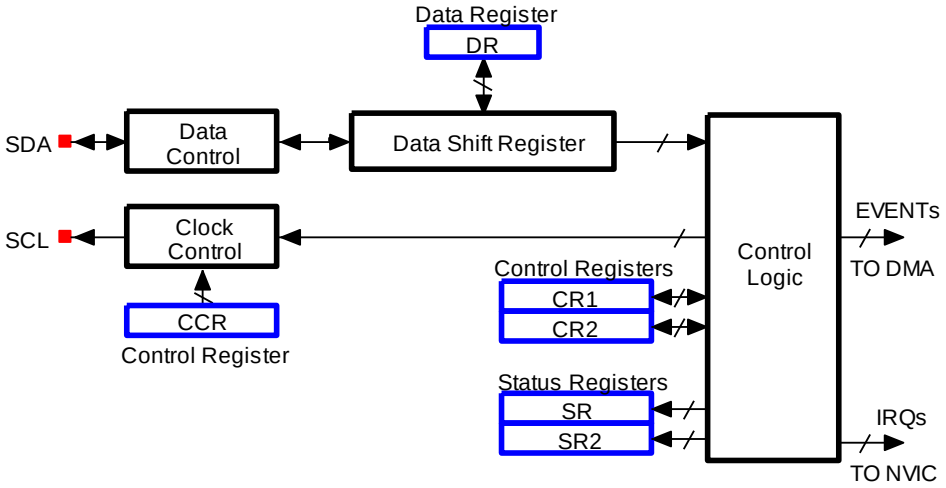


Figure 17.4: A simplified block diagram of an I²C block

The content of the register DR can be accessed by reading from or writing to register ‘I2Cx->DR’. When sending data the processor writes data into the register DR, and the content of this register is copied into the register DSR once the register DSR is free to accept data.

The speed of transmission is defined by control bits in register CCR (Clock Control Register), connected to the Clock Control block. The information of clock pulses, status of SDA and SCL lines, the status of the data register and alike is available to the Control Logic, and can be read-out through status registers SR and SR2. The operation of the control logic is defined by bits in control registers CR1 and CR2. The control logic can issue events and interrupt requests to call assistance to the block. In order to configure these registers efficiently a CMSIS library includes several functions, and they are available in the source file "stm32f4xx_I2c.c". Brief instructions on the use of functions are included in the file. These CMSIS functions require the use of data structures and definitions that are declared in the header file "stm32f4xx_i2c.h".

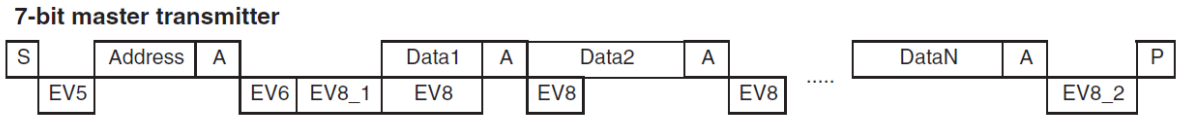
17.2. The I2C protocol – sequence of events, writing

The protocol I^2C requires interaction of the I^2C block with the processor. After every major step in communication the processor must instruct the block on successive action, and all major states of the block are signaled by events. The procedure is as follows: the processor first configures the I^2C block, and then requests a certain action. It takes some time for the block to complete the action, and the processor must wait for the competition which is signaled by an event. Put into hardware: an event toggles a bit in status register SR; when set, the event has happened. The event can be used to trigger an interrupt, or the processor can simply waste time executing an empty loop waiting for the event (the bit to be set). This second option is not very effective, but will be used here for simplicity.

A slave unit commonly houses more than one byte for data, so the master should tell where inside the slave the data should be written to. This is resolved by master sending three bytes; the first byte represents the address of the slave, the second byte represents the address of the register within the slave, and the third byte is the actual data to be written into the selected register within the slave. The procedure for sending a byte from processor into the peripheral using I^2C bus can be dissected into:

1. start the transmission (S),
2. wait for the start condition on the I^2C bus (this is called EV5 in CMSIS for STM32F4xx processors),
3. send the address of the slave (Address) to register DR,
4. wait for the acknowledge signal from the slave (EV6),
5. send the address within the slave (Data1) to register DR,
6. wait for the register DR to become empty (EV8),
7. send the actual data (Data2) to register DR,
8. wait for the acknowledge signal from the slave(EV8_2, here the transmission has finished),
9. stop the transmission (P).

These steps are depicted in Fig. 17.5, a copy from the RM0090, page 826, figure 243 for actions and events during the writing a byte from master to slave, 7 bit addressing.



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge,
EVx= Event (with interrupt if ITEVFEN = 1)

EV5: SB=1, cleared by reading SR1 register followed by writing DR register with Address.
EV6: ADDR=1, cleared by reading SR1 register followed by reading SR2.
EV8_1: TxE=1, shift register empty, data register empty, write Data1 in DR.
EV8: TxE=1, shift register not empty,.data register empty, cleared by writing DR register
EV8_2: TxE=1, BTF = 1, Program Stop request. TxE and BTF are cleared by hardware by the Stop condition

Figure 17.5: Commands and events during the transmission of a byte from master to slave

As stated above: events are connected with bits in status register SR. Waiting for the event EV5, for instance, means waiting for a certain combination of bits in register SR, or, put into the CMSIS terms, waiting for 'I2C_EVENT_MASTER_MODE_SELECT'. Similarly, waiting for event EV6 becomes waiting for 'I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED', and waiting for event EV8 equals waiting for 'I2C_EVENT_MASTER_BYTE_TRANSMITTING'. Finally, waiting for event EV8_2 is the same as waiting for'I2C_EVENT_MASTER_BYTE_TRANSMITTED', see the include file "stm32f4xx_i2c.h", lines 320 to 407, for the justification of terms.

All passing of data and checking of events is also translated into CMSIS terms, and the function "I2C_Write1Byte" to send one byte of data into the slave to a given address is presented next.

```
char I2C_Write1Byte(char SlaveAddr, char addr, char byte1)   {
  while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY));                    // wait if I2C1 is busy   // 2
  I2C_GenerateSTART(I2C1, ENABLE);                                  // generate START         // 3
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));       // wait for EV5           // 4
  I2C_Send7bitAddress(I2C1, (SlaveAddr << 1), I2C_Direction_Transmitter); // send slave addr   // 5
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED)); // wait for EV6    // 6
  I2C_SendData(I2C1, addr);                                         // addr within slave      // 7
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTING));  // wait for EV8          // 8
  I2C_SendData(I2C1, byte1);                                        // send data to slave     // 9
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));   // wait for EV8_2        // 10
  I2C_GenerateSTOP(I2C1, ENABLE);                                   // generate STOP          // 11
  return (0);
}
```

The function receives three arguments for defining the address of the slave ('SlaveAddr'), selecting the register within the slave ('addr') and the data to be sent to this register ('byte1'); all arguments are 8 bits wide and declared as characters. The rest of the function strictly follows what was written in the paragraph above, but in CMSIS terms.

The availability of the I^2C block is tested first in line 2. If the block is busy then the processor waits. The actual procedure starts in line 3 where a processor requests a "start" condition on the I^2C bus, a CMSIS function "I2C_GenerateSTART()" is used. The processor must then wait for the actual "start" condition to be established, and this is done in line 4, where the processor periodically checks the status register against the state 'I2C_EVENT_MASTER_MODE_SELECT' (EV5). The CMSIS function "I2C_CheckEvent()" is used to do the checking.

Once the condition is confirmed the processor proceeds with writing the address of the slave into the register DR, and this is performed by a call to next CMSIS function "I2C_Send7bitAddress()" in line 5. The arguments of the function are the slave address as it should show in the timing diagram from Fig. 17.3 top, followed by a bit to define the read/write option ('I2C_Direction_Transmitter' in this case). The processor must wait for event EV6 now, and this is done in line 6, where the status register is periodically checked against status 'I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED'. Once this status is confirmed it is also true that the slave device responded, and that the processor can continue by sending the address within the slave.

The address within the slave is sent in line 7, and a CMSIS function "I2C_SendData()" is used. The argument is the address within the slave, and the processor must now wait until the register DR is ready to receive next byte; the processor must wait for event EV8. This is performed in line 8 of the function, where the content of the status register is periodically checked against status 'I2C_EVENT_MASTER_BYTE_TRANSMITTING'. Once this status is detected, the processor proceeds to line 9 to send the actual data to the slave.

The data is sent to the slave by the same CMSIS function call, this time the argument is the byte to be sent. This terminates the transmission, and the "stop" condition should be re-established to release the I^2C bus, but not immediately. The processor must wait for all bits to be safely transmitted to the

slave and for slave to confirm the receipt; the processor must wait for event EV8_2, in CMSIS terms 'I2C_EVENT_MASTER_BYTE_TRANSMITTED'. This is performed in line 10, then the "stop" condition is requested in line 11, and the transmission is terminated.

If more than one byte is to be sent to the slave, then commands from lines 8 and 9 should be repeated for each additional byte to be sent, and these bytes are written to consecutive locations within the slave.

17.3. The I2C protocol – sequence of events, reading

A typical reading from slave is performed in two major steps due to properties of most of the slaves. The first major step A is used to send the address within the slave from where the master is supposed to read. The second major step B is used to actually read data from the slave. The procedure for reading a byte from slave to the master using I^2C bus can be dissected into:

A1. start the transmission (S),

A2. wait for the start condition on the I^2C bus (EV5),

A3. send the address of the slave (Address) to register DR, select writing

A4. wait for the acknowledge signal from the slave (EV6),

A5. send the address within the slave (Data1) to register DR,

A6. wait for the acknowledge signal (EV8_2) from the slave telling that the transmission has ended,

B1. start the transmission once again (S),

B2. wait for the start condition on the I^2C bus (EV5),

B3. send the address of the slave (Address) to register DR once more, but select reading,

B4. wait for the acknowledge signal from the slave (EV6),

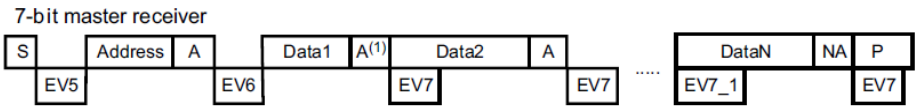
B5. configure the I2C block not to acknowledge the byte received, the reading automatically starts,

B6. wait for one byte to accumulate in the data register (EV7),

B8. retrieve the byte read from the slave by reading it from data register DR,

B9. stop the transmission (P).

Steps under A are the same as for writing and are taken from the previous figure. Steps for B are depicted in Fig. 17.5, a copy from the RM0090, page 828, figure 244 for actions and events during reading a byte from slave to master, 7 bit addressing. Note that reading of a byte in step B5 starts automatically after the configuration of the acknowledge action.



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge, NA= Non-acknowledge,
EVx= Event (with interrupt if ITEVFEN=1)
EV5: SB=1, cleared by reading SR1 register followed by writing DR register.
EV6: ADDR=1, cleared by reading SR1 register followed by reading SR2. In 10-bit master receiver mode, this sequence should be followed by writing CR2 with START = 1.
In case of the reception of 1 byte, the Acknowledge disable must be performed during EV6 event, i.e. before clearing ADDR flag.
EV7: RxNE=1 cleared by reading DR register.
EV7_1: RxNE=1 cleared by reading DR register, program ACK=0 and STOP request
EV9: ADD10=1, cleared by reading SR1 register followed by writing DR register.

Figure 17.5: Commands and events during the reading of a byte from slave to master

Events are again associated with some predefined names in CMSIS library: waiting for the event EV7 means waiting for 'I2C_EVENT_MASTER_BYTE_RECEIVED'.

All passing of data and checking of events is translated into CMSIS terms, and the function "I2C_Read1Byte" to read one byte of data from slave is presented next.

```
char I2C_Read1Byte(char SlaveAddr, char addr)     {
  while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY));                    // wait if I2C1 is busy   // 2
  I2C_GenerateSTART(I2C1, ENABLE);                                  // generate START         // 3
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));       // wait EV5               // 4
  I2C_Send7bitAddress(I2C1, (SlaveAddr << 1), I2C_Direction_Transmitter);  // send device addr // 5
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED)); // wait EV6        // 6
  I2C_SendData(I2C1, addr);                                         // addr within device     // 7
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));  // wait for EV8_2         // 8

  I2C_GenerateSTART(I2C1, ENABLE);                                  // generate START         // 10
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));       // wait EV5               // 11
  I2C_Send7bitAddress(I2C1, (SlaveAddr << 1), I2C_Direction_Receiver);  // send device addr    // 12
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED)); // wait EV6           // 13
  I2C_AcknowledgeConfig(I2C1, DISABLE);                             // do not acknowledge     // 14
  while( !I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_RECEIVED) );   // wait EV7               // 15
  char byte1 = I2C_ReceiveData(I2C1);                               // retrieve byte          // 16
  I2C_GenerateSTOP(I2C1, ENABLE);                                   // generate STOP          // 17
  return (byte1);                                                                            // 18
}
```

The function receives two arguments for defining the address of the slave ('SlaveAddr'), and selecting the register within the slave ('addr'); both arguments are 8 bits wide and declared as characters. The rest of the function strictly follows what was written in the paragraph above, but in CMSIS terms. The function returns a byte retrieved from the slave.

Lines 2 to 7 are the same as used for writing of a byte to the slave, and will not be discussed here again. Line 8 differs, since the processor must wait until a complete address is safely transferred into the slave (EV8_2) before proceeding with steps for reading.

Reading starts in line 10, where the processor requests the "start" condition once again. It must wait for this condition to be implemented, this is done in line 11. Line 12 re-sends the address of the slave, this time the direction is reversed from before. The processor must wait for this address to reach the slave and the slave to acknowledge the receipt of the address, in CMSIS terms it must wait for condition 'I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED'.

The processor then proceeds to initiate the actual reading. This is done by configuring the action of the master whether to acknowledge the receipt of the byte read from the slave or not in line 14. In out example we are reading one byte only, and we should not confirm the receipt in order for the slave to stop sending after one byte. Once the byte is safely stored in the register DR, the condition 'I2C_EVENT_MASTER_BYTE_RECEIVED' can be detected in line 15, and the processor can read the content of the register DR in line 16. Since this is the last byte to be read, the operation is terminated by requesting a "stop" condition in line 17 and returning the byte read in line 18.

If more than one byte is to be read from the slave, then the last part of the function following line 13 should be replaced by the listing below.

```
  I2C_AcknowledgeConfig(I2C1, ENABLE);                              // acknowledge 1st byte   // 14
  while( !I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_RECEIVED) );   // wait EV7               // 15
  char byte1 = I2C_ReceiveData(I2C1);                               // retrieve byte          // 16
  I2C_AcknowledgeConfig(I2C1, DISABLE);                             // do not acknowledge 2nd byte
  while( !I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_RECEIVED) );   // wait EV7               // 18
  char byte2 = I2C_ReceiveData(I2C1);                               // retrieve byte          // 19
  I2C_GenerateSTOP(I2C1, ENABLE);                                   // generate STOP          // 20
  return ((byte2 << 8) + byte1);                                                             // 21
```

The master should acknowledge the receipt of the first byte from the slave, and not acknowledge the receipt of the second byte. Both bytes are combined into a 16 bit short integer and returned; this requires a redefinition of the function as well from ‘char’ to ‘short’.

17.4. The program to read data from sensor LIS3LV02DQ using I²C bus

The program to initialize and read the value from the accelerator chip is listed next.

```
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_i2c.c"
#include "LCD2x16.c"
#include "dd.h"

#define LIS_chip_I2C_address 0x1d

int main ()  {
  I2C1init();                                  // I2C1 init
  LCD_init();                                  // Init LCD
  LCD_string("I2C demo Z", 0);                 // write title
  LCD_string("X        Y", 0x40);              // write title

  I2C_Write1Byte (LIS_chip_I2C_address, 0x20,0xc7);                                    // 15

  while (1) {
    LCD_sInt3DG(I2C_Read2Byte (LIS_chip_I2C_address, 0x28), 0x41, 0x01);      // show acc x
    LCD_sInt4DG(I2C_Read2Byte (LIS_chip_I2C_address, 0x2a), 0x4a, 0x01);      // show acc y
    LCD_sInt4DG(I2C_Read2Byte (LIS_chip_I2C_address, 0x2c), 0x0a, 0x01);      // show acc z
    for (int i = 0; i < 10000000; i++)   {};    // waste time
  };
}
```

The listing starts with inclusion of necessary files, and defines the address of the slave chip as 0x1d, as specified in the datasheet of the chip LIS3LV02DQ, the accelerometer. The processor starts the execution at “main”, where it first calls the configuration function “I2C1init()”, to be discussed still. The execution continues with a call to configuration function for LCD screen and two introductory writes to the screen.

The call in line 15 configures (turns on for all three axes) the accelerometer chip, as described in its datasheets, by writing a value 0xc7 into location 0x20.

The execution of the program then enters the infinite loop where three 16 bit numbers (the acceleration in all three axes) are retrieved from the accelerometer chip and displayed at the LCD.

The last thing to discuss is the configuration of the microcontroller and the I²C block. We are going to use I²C block I2C1, its connections are available at port B, pins 6 and 7. The configuration includes mapping of port pins to the I²C block, and the configuration of I²C block itself. The function is listed below.

```
void I2C1init(void) {
GPIO_InitTypeDef       GPIO_InitStructure;
I2C_InitTypeDef        I2C_InitStructure;

  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);                                // 5
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1,  ENABLE);                                // 6

  GPIOB->BSRRL = BIT_6 | BIT_7;                          // SDA, SCL -> hi            // 8
  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_6 | GPIO_Pin_7; // SDA, SCL def            // 9
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AF;          // alternate function        // 10
```

```
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;                                    // 11
  GPIO_InitStructure.GPIO_OType = GPIO_OType_OD;          // use open drain !          // 12
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_25MHz;                                    // 13
  GPIO_Init(GPIOB, &GPIO_InitStructure);                                               // 14

  GPIO_PinAFConfig(GPIOB, GPIO_PinSource6, GPIO_AF_I2C1);  // PB6:I2C1_SCL            // 16
  GPIO_PinAFConfig(GPIOB, GPIO_PinSource7, GPIO_AF_I2C1);  // PB7:I2C1_SDA            // 17

  I2C_InitStructure.I2C_ClockSpeed      = 100000;                                      // 19
  I2C_InitStructure.I2C_Mode            = I2C_Mode_I2C;                                // 20
  I2C_InitStructure.I2C_DutyCycle       = I2C_DutyCycle_16_9;                          // 21
  I2C_InitStructure.I2C_OwnAddress1     = 0;                                           // 22
  I2C_InitStructure.I2C_Ack             = I2C_Ack_Disable;                             // 23
  I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;            // 24
  I2C_Init(I2C1, &I2C_InitStructure);                                                  // 25
  I2C_Cmd(I2C1, ENABLE);                                                               // 26
}
```

Two data structures are required by CMSIS functions for configuration, those are declared first. Next clock signals for port B and I2C1 blocks are enabled in lines 5 and 6.

The configuration of port B is given in lines 8 to 17. Pins 6 and 7 are to be dealt with, those are indicated in line 9. Both pins should implement alternate function, have no terminating resistors, and use open drain configuration of the output, as stated in lines 10 to 12. The speed of operation is not critical and selected as 25MHz here. The configuration function for port “GPIO_Init()” is called in line 14 utilizing the pointer to the initialized data structure ‘GPIO_InitStructure’.

The alternate function is selected next for those two pins in lines 16 and 17.

Lines 19 to 26 are used to initialize and configure the I2C1 block. A brief description of the configuration procedure is given in the source file “stm32F4xx_I2C.c”. The data structure ‘I2C_InitStructure’ is described in the header file “stm32f4xx_i2c.h”, lines 54 to 73, and is initialized as follows:

- The member ‘.I2C_ClockSpeed’ specifies the clock frequency for the data transfer, it must be less than 400000 [Hz].
- The member ‘.I2C_Mode’ specifies the mode of operation for the I^2C block; the block can implement either I^2C protocol or SM bus protocol. In our example the I^2C protocol is required, and the member is initialized accordingly.
- The member ‘.I2C_DutyCycle’ specifies the ratio of clock pulse width versus the pause between pulses; the possibilities are listed in the header file in lines 114 and 115, and are of no importance here; an arbitrary one of the available is selected.
- The member ‘.I2COwnAddress1’ specifies the first device own address, in our case 0.
- The member ‘.I2C_Ack’ enables or disables the acknowledgement step in general; we will disable it here, and enable it when needed by appropriate CMSIS function call.
- The last member ‘.I2C_AckknowledgeAddress’ specifies the expected length of the address; it can be either 7 or 10 bit, and is initialized to 7 bit in our example.

The function call in line 25 uses this data structure to configure the I2C1 block, and the last function call in line 26 enables the I2C1 block.

18.Serial communication - SPI

Some sensors implement SPI (Serial Peripheral Interface) protocol for data transfer. An example of communication between a microcontroller and an accelerometer sensor using the SPI interface will be demonstrated in this example.

18.1. The SPI protocol and hardware

The SPI protocol defines a bus with four wires (four signals) and a common ground. There is one master device controlling the activity on the bus, and one active slave device. The slave is active only when the signal Slave Select (SS) enables it; this signal is provided by the master. There can be more than one slave connected to the SPI bus, but each slave requires its own Slave Select (SS1, SS2, …) signal, see Fig. 18.1. Data gets transferred serially bit-by-bit. There are two signals to carry information, one from master to slave (MOSI, Master Output Slave Input, driven by master), and one for the opposite direction (MISO, Master Input Slave Output, driven by slave). The last signal SCLK (Serial CLocK) assures the time synchronization between master and slave, and is always driven by master. There are streamlined versions of the SPI bus using only one signal to transfer data, but the direction of data must be reversed on request; we will not use this kind of data transfer in this example.

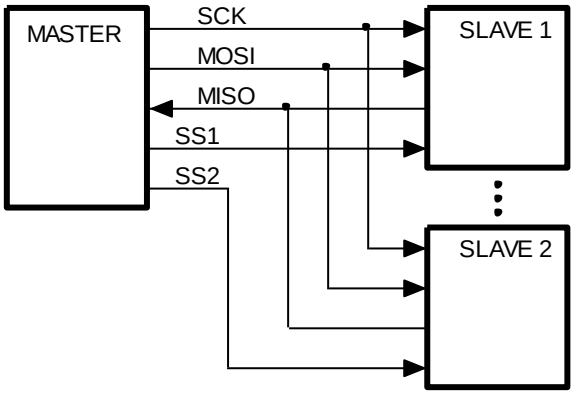


Fig. 18.1: The SPI bus can be used to transfer data between the master and the slave using four wires.

The speed of data transfer is higher than with I^2C bus, since the slave is selected using a hardware signal and there is no need to transfer the address of the slave. However, this requires multiple Slave Select signals if more than one slave is connected to the bus, and therefore more pins at the microcontroller. The speed of transfer can also be higher due to the device outputs which are able to force signals low or high, contrary to the open-drain outputs used at I^2C which can force signals low only. The logic levels are depend on the power supply for the devices connected to the SPI bus. The achievable speed of transmission conforms to the slowest device on the bus, as with I^2C bus.

The SPI protocol is less strict that the I2C protocol due to the fact that it was implemented first by several different companies and standardized only later. Variants of clock polarities, edge synchronizations, and even number of bits per transfer are used, and the designer should adopt its hardware to the SPI devices used. The microcontroller used here can implement some possible variants of the standard, and the variant implemented in the accelerometer LIS3LV02DQ is one of them.

The timing diagram of the required signals for the communication with the accelerometer chip is given in Fig. 18.2, the writing from master to slave being shown in the upper half. Slave select signal SS must first be forced low by the master, then a series of 2 times eight clock pulses (low-to-high

transitions) are issued by the master at the SCK line. After this the signal SCK stays high, followed by the signal slave select SS. The value of the signal MOSI is clocked into the slave on positive edge of the clock signal SCK, and the MOSI signal can change either before or after the clocking edge, see the datasheet on LIS3LV02DQ for details. The first eight bits start with a bit to define either writing to slave (low) or reading from slave (high); this bit is low here. The next bit is fixed to zero, followed by six bits of address. This is the address to be used by the slave device to select one of its internal locations for writing, not the address of the slave on the bus, as with I^2C bus. Here, with SPI bus, the device is selected using the Slave Select signal! Next eight bits are simply the byte as it is supposed to be written into the slave. In the diagram address 0x21 is selected for writing, and value 0x40 is written. The signal MISO is not used during writing, and is not shown.

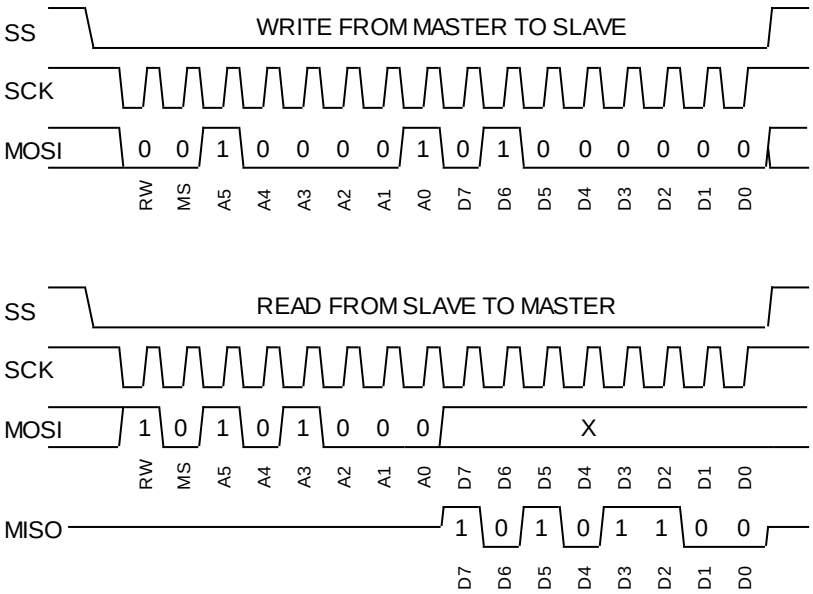


Figure 18.2: SPI bus signals during the writing into the slave (top), and reading from the slave (bottom)

The timing diagram for reading is given in the same figure, bottom half. The Slave Select signal and the 16 rising edges of the clock are the same as for writing, signals MOSI and MISO differ. The master first sends a command to read from the slave by forcing the first bit of the first byte at MOSI line high, and then pulling the same line low during the second bit. Next come six bits of address within the slave, here 0x28. After the first byte the MOSI signal is not important anymore. It can be left floating as shown in the figure, but it can also have any other logic value; its value will be ignored by the slave. However, clock pulses are still coming, and the slave now drives the MISO signal returning the byte to be read, shown as 0xAC. This sequence should be accepted by the SPI block within the microcontroller and combined into a byte.

18.2. The SPI protocol - writing

The Discovery board exposes pins associated with SPI block 2, making them available to the user. Additionally, the BaseBoard provides two connectors for SPI slaves both having separate, hardware defined Slave Select signals; the one we will use in this example is taken from port B, pin 12. Signals SCK, MOSI and MISO are available as alternate function at pins 13 to 15. It is convenient to summarize the steps for writing into:

1. Enable the Slave Select signal (port B, pin 12) by pulling it low.
2. Wait some time.
3. Send address and data from master to the slave using the SPI2 block; this is done by a call to a CMSIS function “SPI_I2S_SendData()”, which simply writes 16 bits into the data register DR of

the SPI2 block. Do not miss to keep the MSB of the 16 bit low, since this indicates “write” to the slave. The CMSIS function can be replaced by a hard-coded write into register SPI2->DR, as shown in this example.

4. Wait some time for all bits to be transferred to slave.
5. Disable the Slave Select signal (port B, pin 12) by pulling it high.

These steps are implemented in the function “LIS_write()” below:

```
void LIS_write (char address, char data)        {
  GPIOB->BSRRH = BIT_12;                                                    // step 1
  for (int i = 0; i < 150; i++)   {};           // waste time               // step 2
  SPI2->DR = ((unsigned short)(address & 0x3f) << 8) + (unsigned short)(data);         // step 3
  for (int i = 0; i < 1200; i++)  {};           // waste time               // step 4
  GPIOB->BSRRL = BIT_12;                                                    // step 5
}
```

Step 3 here may need additional discussion. The address must be a six bit value. However, since the ‘address’ is an argument to this function, an unwary user might specify more than six bits, and it is best to clip it to six bits by AND-ing it with 0x3f. This address must come to upper eight positions of the data register, so the software converts is into unsigned short integer, and shifts it for eight places. Finally, the ‘data’ is added to lower eight positions of the final value to be written into the ‘SPI2->DR’.

The waiting in steps 2 and 4 is done by an empty loop, and is not very efficient and precise.

18.3. The SPI protocol – reading

Similarly, the steps required for reading from the slave can be summarized into:

1. Enable the Slave Select signal (port B, pin 12) by pulling it low.
2. Wait some time.
3. Send address from master to slave using the SPI2 block, the address must occupy the upper 8 bits of the short integer passed to a CMSIS function call “SPI_I2S_SendData()”, which simply writes 16 bits into the data register DR of the SPI2 block. Do not miss to keep the MSB of the 16 bit high, since this indicates to read from the slave. The CMSIS function can be avoided by a hard-coded write into the register SPI2->DR.
4. Wait some time for all bits to be transferred to slave.
5. Disable the Slave Select signal (port B, pin 12) by pulling it high.
6. Read the content of the data register SPI2->DR in the SPI2 block, it contains the value returned by the slave. A CMSIS function “SPI_I2S_ReceiveData()” could be used instead of the hard-coded read from the register.

These steps are implemented in the function “LIS_read()” below:

```
short LIS_read (char address) {
  GPIOB->BSRRH = BIT_12;                                                    // step 1
  for (int i = 0; i < 150; i++)   {};           // waste time               // step 2
  SPI2->DR = ((unsigned short )(address & 0x3f) << 8) + 0x8000;             // step 3
  for (int i = 0; i < 1200; i++)  {};           // waste time               // step 4
  GPIOB->BSRRL = BIT_12;                                                    // step 5
  return (SPI2->DR & 0xff);                                                 // step 6
}
```

The step 3 might need additional discussion. The address of the register within the SPI slave must be reside in the upper eight bits of the unsigned short integer to be written into the register SPI2-DR, and this is accomplished by shifting it to the left for eight places. However, the MSB of the shifted value must he high to signal reading from the SPI slave, and this is achieved by OR-ing the result of shifting with 0x8000.

The value obtained from the SPI slave is available in the SPI2->DR register, and only lower eight bits are to be used; those are isolated by an AND function and argument of 0xff.

Once we have functions to write and read the SPI bus we are ready to see the program to communicate with the accelerometer and pass results on the screen. The listing is given below.

```
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_spi.c"
#include "LCD2x16.c"
#include "dd.h"

int main ()  {
  SPI2init();                                  // SPI2 init                                  // 8
  LCD_init();                                  // Init LCD
  LCD_string("SPI demo Z", 0);                 // write title
  LCD_string("X        Y", 0x40);              // write title

  LIS_write (0x20,0xc7);                                                                  // 13

  while (1) {
    short xLSB = LIS_read(0x28); short xMSB = LIS_read(0x29) << 8;                        // 16
    short yLSB = LIS_read(0x2a); short yMSB = LIS_read(0x2b) << 8;                        // 17
    short zLSB = LIS_read(0x2c); short zMSB = LIS_read(0x2d) << 8;                        // 18
    LCD_sInt16((xLSB + xMSB), 0x41, 0x01);     // show acc x                              // 19
    LCD_sInt16((yLSB + yMSB), 0x4a, 0x01);     // show acc y                              // 20
    LCD_sInt16((zLSB + zMSB), 0x0a, 0x01);     // show acc z                              // 21
    for (int i = 0; i < 10000000; i++)   {};   // waste time
  };
}
```

The listing starts with the inclusion of necessary source files, and the execution starts with the function “main()”. A function to configure the SPI2 block is called in line 8, the content of this function will be discussed last. The function for the configuration and initialization of the LCD screen is called next, followed by two commands to write introductory strings to the LCD.

The accelerometer sensor is turned on and configured by a single write in line 13. As in the previous chapter: value 0xc7 must be written to sensor at address 0x20 in order for sensor to operate.

The execution proceeds with the loop, where the content of six registers within the sensor are read periodically using the derived function for reading the SPI bus, these are lines 16 to 18. Next the results obtained are combined and written to the LCD screen at appropriate positions.

The loop terminates with an empty ‘for’ loop to waste some time and to reduce the frequency of results being written to the LCD.

Lines 16 to 18 need some discussion. The datasheet of the accelerometer specifies that the result of the measurement is available in registers 0x28 (LSB) and 0x29 (MSB) for the acceleration in x-axis. Registers 0x2a (LSB) and 0x2b (MSB) hold the result for y-axis, and registers 0x2c (LSB) and 0x2d (MSB) hold the result for z-axis. These bytes are read from the relevant registers one by one and stored in six short integers ‘xLSB’, ‘xMSB’, ‘yLSB’... These integers are then combined to form complete results for accelerations in each of the three axes.

The configuration procedure for the SPI block 2 is discussed last. Its listing is given below, and it consists of two parts. The main part is called “SPI2init()”, and is used to configure the SPI block. However, the SPI signals must be assigned to pins on the microprocessor, and this is done in the function “GPIOBinit_SPI2()” which is called from the “SPI2init()”.

```
void GPIOBinit_SPI2(void)   {
GPIO_InitTypeDef          GPIO_InitStructure;
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);                                   // 3
```

```
  GPIOB->BSRRL = BIT_14 | BIT_13 | BIT_12 | BIT_11;                                    // 5

  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;            // 7
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AF;                                       // 8
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;                                   // 9
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;                                      // 10
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_25MHz;                                   // 11
  GPIO_Init(GPIOB, &GPIO_InitStructure);                                              // 12

  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_11 | GPIO_Pin_12; // PB12, PB12: SPI_SEL   // 14
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_OUT;                                      // 15
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_UP;                                       // 16
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;                                      // 17
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_25MHz;                                   // 18
  GPIO_Init(GPIOB, &GPIO_InitStructure);                                              // 19

  GPIO_PinAFConfig(GPIOB, GPIO_PinSource13, GPIO_AF_SPI2);  // PB13:SPI2_SCK         // 21
  GPIO_PinAFConfig(GPIOB, GPIO_PinSource14, GPIO_AF_SPI2);  // PB14:SPI2_MISO        // 22
  GPIO_PinAFConfig(GPIOB, GPIO_PinSource15, GPIO_AF_SPI2);  // PB15:SPI2_MOSI        // 23
}

void SPI2init(void) {       //
SPI_InitTypeDef   SPI_InitStructure;
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2,  ENABLE);                               // 3
  GPIOBinit_SPI2();     // GPIO clock enable, digital pin definitions                 // 4

  SPI_InitStructure.SPI_Direction       = SPI_Direction_2Lines_FullDuplex;            // 6
  SPI_InitStructure.SPI_Mode            = SPI_Mode_Master;                            // 7
  SPI_InitStructure.SPI_DataSize        = SPI_DataSize_16b;                           // 8
  SPI_InitStructure.SPI_CPOL            = SPI_CPOL_High;                              // 9
  SPI_InitStructure.SPI_CPHA            = SPI_CPHA_2Edge;                             // 10
  SPI_InitStructure.SPI_NSS             = SPI_NSS_Soft;                               // 11
  SPI_InitStructure.SPI_BaudRatePrescaler       = SPI_BaudRatePrescaler_64;           // 12
  SPI_InitStructure.SPI_FirstBit        = SPI_FirstBit_MSB;                           // 13
  SPI_InitStructure.SPI_CRCPolynomial   = 7;                                          // 14
  SPI_Init(SPI2, &SPI_InitStructure);                                                 // 15

  SPI_Cmd(SPI2, ENABLE);                                                              // 17
}
```

The configuration of the SPI block is initiated by the call of the function “SPI2init()”. The function “SPI_Init()” from a CMSIS library will be used for the configuration as described in the source file “stm32f4xx_spi.c”. This function requires the use of the data structure ‘SPI_InitStructure’, which is defined in the header file “stm324xx_spi.h”, lines 54 to 85.

The clock for the block SPI2 is enabled in line 3, then the function that deals with pin configuration at port B is called; this will be discussed later. The data structure is initialized in lines 6 to 14 as follows.

- Member ‘.SPI_Direction’ defines the number of wires used for the transmission and the direction of data transfer, see the Reference manual RM0090, pg. 863 for details. We are going to use three lines, one for the clock signal, one to send data from the microcontroller to the accelerometer chip, and one to receive data into the microcontroller from the accelerometer chip. The synonymous for this in CMSIS is the keyword ‘SPI_Direction_2Lines_FullDuplex’. Alternative keywords are listed in the header file, lines 151 to 154.

- Member '.SPI_Mode' defines the mode of operation for the block SPI; it can be either master or slave. In our case the microcontroller is taking the complete control over the SPI bus, and this is accomplished by making the SPI block the master of the bus.
- Member '.SPI_DataSize' defines the length of the message sent in one package. It can be either 8 or 16 bits, here we opt for 16 bits by initializing the member with the keyword 'SPI_DataSize_16b'.
- Members '.SPI_CPOL' and '.SPI_CPHA' define the relation of the polarity of the clock signal SCL and the edge of this signal to be used for clocking of data, see the details in the Reference manual RM0090, pages 857 and 858. The sensor used in our experiment requires signals as given in Fig. 18.2, and this is accomplished by initializing these two members as 'SPI_CPOL_High' and 'SPI_CPHA_2Edge'.
- Member '.SPI_NSS' defines the generation of the slave select signal. We intend to generate this signal by software, and this is accomplished by initializing this member with 'SPI_NSS_Soft'.
- Member '.SPI_BaudRatePrescaler' defines the speed of transmission. We arbitrary select one of the division ratios in this experiment as 'SPI_BaudRatePrescaler_64'. Note that the slave select signal is managed by software and a software delay is used to align the select signal. Using different value for the prescaler requires the change in the delay loops as well.
- Member '.SPI_FirstBit' defines the order the bits are sent to the slave. The chain can start with the MSB or the LSB, and the accelerometer chip used in this experiment requires MSB to be sent first.
- Member '.SPI_CRCPolynomial' is not used here since we do not use any error checking. Nevertheless, this member is initialized to 7.

Following the initialization of data structure the block gets configured by a call in line 15. The configuration is followed by a call in line 17 that enables the block SPI for the use.

The complete list of special functions that can be assigned to pins is given in table 9, pg. 60 and on, in the datasheet of the microcontroller (DM00037051). Following this table the SPI signals for block 2 are available at port B, pins 12 to 15; these pins need to be configured before use, and this is done in function "GPIOBinit_SPI2". The function starts with the definition of the data structure and enabling of the clock for port B. Since the initial state of all lines is supposed to be high all pins are initialized to logic high in line 5.

Pins 13, 14 and 15 are used for SPI signals SCK, MISO and MOSI respectively. These pins are designated as outputs (lines 7 to 12), and alternate functions are selected for them (lines 21 to 23). The rest of the initialization of the data structure and function call itself follow the usual pattern for pin configuration.

There are two slave select pins 11 and 12; pin 12 is used in this example since only one of the SPI connectors is utilized. These pins are designated as outputs in lines 14 to 19.

Finally the alternate function for pins 13 to 15 is selected in lines 21 to 23.

19. FIR filtering

Digital filtering of analog signals in real time is easy to implement when one has an ADC, a processor, and a DAC, Fig. 19.1. An example of FIR (Finite impulse Response) filtering will be given.

Figure 19.1: The blocks involved in FIR filtering.

19.1. The theory of FIR filtering

The FIR (Finite Impulse Response) filtering implements a convolution of input signal with predefined coefficients to achieve filtering effect. The convolution formula is given by ([1], chapter 6):

$$y_k = \sum_{m=0}^{M-1} h_m x_{k-m}$$

The coefficients h_m define the properties of the filter, and are calculated as the inverse Fourier transform of the frequency response for the desired filter. Coefficients h_m span for positive and negative indexes m, from –M to +M (theoretically M approaches infinity, but is truncated in practice). This effectively means that in order to calculate the filter response at time k, one should know samples x from current time k, from past (x_{k-1}, x_{k-2}, … , x_{k-m}), and also from the future (x_{k+1}, x_{k+2}, … , x_{k+M}). Since knowing of the future is not the privilege of ordinary people (or processors), we cannot implement convolution formula directly, but have to resort to a delayed calculation by modifying/rearranging the convolution formula:

$$y_k = \sum_{m=-M}^{M} h_m x_{k-m-M}$$

This is graphically presented in Fig. 19.2. Input samples *x* are stored in a (circular) buffer one per box in the drawing, and samples from the past with indexes from $k - 2M$ to k are used to calculate convolution. The result y_k used as current output from the filter is actually a delayed version of the filtered input signal *x*. The delay depends on the number of coefficients used in convolution.

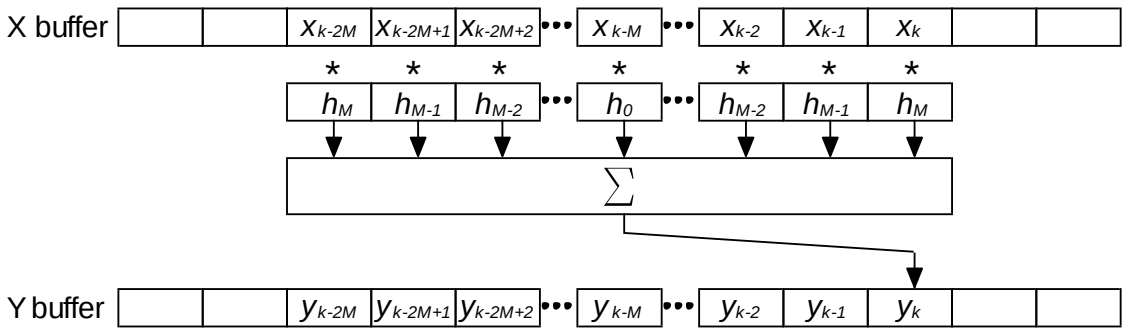


Figure 19.2: The samples used in FIR filtering

The coefficients h_m for a low pass FIR filter are given by (again [1], chapter 6):

$$h_m = 2\frac{f_c}{f_s} \cdot \frac{\sin\left(2\pi m \frac{f_c}{f_s}\right)}{2\pi m \frac{f_c}{f_s}} ,$$

where f_s stands for the sampling frequency, and f_c for the corner frequency of the filter. The sharpness of the filter is better for a large number of coefficients (*M*). However, a large number of coefficients imply many multiplications in the convolution formula, so one has to choose *M* carefully.

The attenuation might not be the expected one due to the limited number of coefficients used, but can be improved by gradually reducing values of coefficients close to index *M*; the process is known as "windowing". A common window function is a raised cosine, named by its inventor von Hann window.

$$h_{m\,windowed} = h_m \cdot \frac{1 + \cos(\pi \cdot \frac{m}{M-1})}{2}$$

19.2. FIR filtering in real time - software

FIR filtering can be implemented in real time. Input signal must be sampled at regular time intervals, and this can be achieved as described in chapter 13. The timer can start the acquisition, and then the sampled value can be stored into a circular buffer *X*. The buffered samples can be used in calculation of the convolution formula following the acquisition of each new sample, and the result of convolution can be converted back to analog signal using a DAC or stored into another circular buffer *Y* for further use. The important thing is that the calculation of convolution must be performed immediately after a new sample of the input signal is available, therefore within the interrupt function. The calculation must be finished before the next interrupt request; this additionally limits the number of coefficients h_m in the convolution formula since time is spent for every multiplication.

Coefficients used in convolution formula stay the same throughout the filtering process, and should be calculated only once prior to the filtering.

The listing of the program is given in Fig. 3. It commences with the inclusion of CMSIS files. Next two circular buffers *X1* and *X2*, and a pointer *k* are declared; these must be global variables since they are used in the interrupt function and must retain their content from one execution of the interrupt function to another. The length of circular buffers is exaggerated, but shows that a long buffer is not a problem for the microcontroller. The buffers are declared as short integers, 16 bit signed values; enough to hold the result from an ADC. Next an array for coefficients *h* is declared, and is composed of "float" values. These coefficients have values of less than one, and must be declared as floating point numbers, apparently. Again, the length of this array is exaggerated.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_adc.c"
#include "stm32f4xx_dac.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_tim.c"
#include "dd.h"
#include "math.h"
#define pi 3.14159

short int X1[1024], X2[1024], k = 0;                                          // 11
float h[1024];                                                                // 12

int main ()  {
  h[0] = 2.0 * 100.0 / 10000.0;         // central weight: 2 x fc / fs       // 15
  for (short m = 1; m < 64; m++)                                              // 16
```

```
    h[m] = (h[0] * sin(pi * m * h[0])) / (pi * m * h[0]);       // other weights              // 17
  for (short m = 1; m < 64; m++)                                                              // 18
    h[m] = (h[m] * cos(pi / 2 * m / 63.0));                     // windowing                  // 19

  GPIOEinit ();                                                                               // 21
  ADCinit_T5_CC1_IRQ();                                                                       // 22
  DACinit();                                                                                  // 23
  TIM5init_TimeBase_CC1(8400);          // 8400 == 100us == 10kHz                             // 24

  while (1) {                           // endless loop                                       // 26
  };
}
```

The body of the program starts in line 15. Coefficients are calculated in lines 15 to 17. As we can see from the formula above the values of coefficients are symmetrical around the central one with index 0, therefore we only need to calculate coefficients with positive indexes *m* and the one with index *m*=0, altogether 64 coefficients. Lines 18 and 19 implement the windowing.

Peripherals are configured and initialized by calls in lines 21 to 24. The functions called are the same as used in previous chapters. They turn on two ADCs and two DACs, define their properties and configure associated port pins. The configuration also starts a timer to issue periodic Start Conversion pulses for the ADC at 100μs time intervals, and enables interrupt requests from the ADC.

The microcontroller is now ready to start filtering and the program continues into an endless loop where it wastes time. The important stuff happens in the interrupt function listed below.

```
void ADC_IRQHandler(void)
{
  GPIOE->BSRRL = BIT_8;
  X1[k] = ADC1->DR;
  X2[k] = ADC2->DR;
  float conv = (float)X1[(k - 100) & 1023] * h[0];                                            // 6
  for (int m = 1; m<64; m++)                                                                  // 7
    conv += h[m] *(float)(X1[(k - 100 + m) & 1023] + X1[(k - 100 - m) & 1023]);               // 8
  DAC->DHR12R1 = (int)conv;                                                                   // 9
  DAC->DHR12R2 = X1[(k - 100) & 1023];                                                        // 10
  k++; k &= 1023;                                                                             // 11
  GPIOE->BSRRH = BIT_8;
}
```

When a new sample is ready in the ADC the interrupt function is called. Results from both ADCs are stored into circular buffers *X1* and *X2* at location pointed to by the pointer *k*. Next a float variable *conv* is declared in line 6, and a product of the central weight *h[0]* and central sample *X1[(k-100) & 1023]* is stored into it. Here we assume that an offset of 100 from the current sample is sufficient to cover the length of the convolution; the offset should be bigger than *M*. The AND function within the square brackets is used to avoid accessing the circular buffer beyond its boundaries as explained in chapter 13.

The rest of the convolution is implemented within the 'for' statement, lines 7 and 8, for coefficients with indexes from 1 to including 63. Coefficient values are symmetrical around the central coefficient, and it would be a waste of time to make two separate multiplications of input samples with the same value of coefficient. It is better to add the two (symmetrical) input samples first, and then multiply the sum by the coefficient. The input samples are indexed as *[k – 100 + m]* and *[k – 100 - m]* to emphasize the symmetry, and the AND function is used to keep the pointer within the circular buffer.

Once the convolution is calculated the result is converted to integer form and sent to the first DAC. The unfiltered original signal is sent to the second DAC for comparison. This signal must be equally delayed as the filtered one; the central sample as used in convolution is sent.

The two GPIOE writes are used to make a pulse at port E, pin 8, and the pulse can be used to determine the time needed to execute the interrupt function (12μs in this case).

The speed of execution can be improved slightly by using integer variables and integer mathematical operations. As stated before the coefficients have values less than one and cannot be converted to integers. However, they can be multiplied by 65536 (which is equivalent for shifting the coefficient values for 16 bits to the left) and then converted to integers. Since we now have integer coefficients and integer samples of input signals all calculations can be performed in integer arithmetic. Since coefficients are 65536 times too big now the convolved results are also65356 times too big, and have to be divided by 65536 (shifted by 16 bits to the right) to obtain the correct value.

Only few lines of program need to be changed to implement the idea. The declaration of the array with coefficients *h[1023]* is changed to "int" in line 11.

The calculation of coefficient values in "main" part of the program is changed. First a value of the central coefficient is calculated as "float", but is immediately converted to integer and stored into the array of coefficients at index 0. Next coefficients for indexes from 1 to 63 are calculated, all multiplied by 65536. In order to implement the windowing the calculated coefficients are again converted to floating point numbers, multiplied by the weight, and converted back to integers, see the listing below.

```
float h0 = 2.0 * 100.0 / 10000.0;
  h[0] = (int)(h0 * 65536);                 // central weight: 2 x fc / fs
  for (short m = 1; m < 64; m++)
    h[m] = (int)((h0 * 65536 * sin(pi * m * h0)) / (pi * m * h0));     // other weights
  for (short m = 1; m < 64; m++)
    h[m] = (int)((float)h[m] * cos(pi / 2 * m / 63.0));                // windowing
```

In the interrupt function the intermediate variable "conv" is declared as integer, and the calculation is performed without explicitly stating the integer arithmetic. This gets done automatically when all variables used are integers. The only difference follows at the point where the result of convolution is sent to the DAC; it first gets divided by 65536 (shift right for 16 bits), and then written to the DAC.

```
  int conv = X1[(k - 100) & 1023] * h[0];
  for (int m = 1; m<64; m++)
    conv += h[m] *(X1[(k - 100 + m) & 1023] + X1[(k - 100 - m) & 1023]);
  DAC->DHR12R1 = X1[(k - 100) & 1023];
  DAC->DHR12R2 = conv >> 16;
```

The execution of the interrupt function using integer arithmetic takes about 9μs.

An interesting FIR function that can be implemented using the FIR filtering is the phase shifting of a signal for 90 degrees; the amplitude of the shifted version remains unchanged. Such FIR function is called the Hilbert transform. The coefficients for a Hilbert transform are given by:

$$h_m = \frac{-1 + (-1)^m}{\pi m}$$

The same program as used for the integer version of FIR filtering can be modified to perform the 90 degree shift, but coefficients must be calculated following the formula above, see listing bellow.

```
  h[0] = 0;
  for (short m = 1; m < 64; m++)
    h[m] = (int)(65536.0 * (-1.0 + cos(pi * m)) / (pi * m));           // coefficients
  for (short m = 1; m < 64; m++)
    h[m] = (int)((float)h[m] * cos(pi / 2 * m / 63.0));                // windowing
```

Note that here also the coefficients are multiplied by 65535, and that the result of convolution must be divided by the same factor.

[1] The Scientist and Engineer's Guide to Digital Signal Processing

20. IIR filtering

The IIR filtering is very similar to FIR filtering as far as the implementation in the microcontroller is concerned. An example of a program for IIR (Infinite Impulse Response) filtering will be given.

20.1. The theory of FIR filtering

Mathematically, the IIR filtering is expressed as:

$$y_k = \sum_{m=0}^{M} a_m x_{k-m} + \sum_{n=1}^{N} b_n y_{k-n}$$

Here coefficients (weights) are marked a_m and b_n, *x* are input samples and *y* is the result of filtering in one step *k*. The coefficients are determined using more complex algorithms than the inverse Fourier transform used for FIR coefficients h_m.

The procedure to calculate the output from a filter is presented in Fig. 19.1. There are two (circular) buffers involved, one for input samples *X* and one for output results *Y*. The new result y_k is composed by adding together two convolutions. The first convolution involves input samples *x* and coefficients *a*, the second convolution involves past results *y* and coefficients *b*. The new result y_k is sent to the DAC as the result of filtering, and also stored to be used in successive calculations.

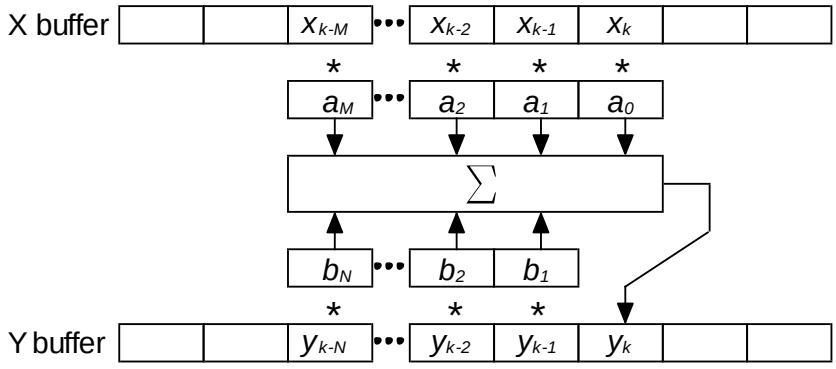


Figure 20.1: Graphical representation of the IIR filtering

The implementation of IIR filtering is based on the previous example on FIR filtering. The listing of the program is given bellow. Functions for the initialization of peripherals are the same as used in chapter on FIR filtering, and are not discussed again.

There are two circular buffers declared as global variables, since they must be reachable from the interrupt function. The buffer for input samples *X* is declared as integer, while buffer for the output results *Y* is declared as float. The reason is the required precision; with IIR filters the numerical errors caused by the use of integer variables may lead to poor filter performance or even to numerical instabilities and oscillations. The output buffer could also be integer, but it should be up-scaled as were the coefficients in the former example on FIR filtering. There are two pairs of such circular buffers *X1*, *X2* and *Y1*, *Y2* to allow two signals to be filtered simultaneously. The lengths of buffers are exaggerated.

The procedure to determine the values of coefficients *a* and *b* can be complex, and is out of the scope of this text. Luckily for us, tables with coefficients exist. For the purpose of this programming example the coefficient values are copied from reference [1]. A fourth order filter with Chebishew characteristics is implemented. The corner frequency of 0.025 of the sampling frequency and 0.5% ripple in the pass-band are used. The coefficients are:

m/n	0	1	2	3	4
a_m	1.504626e-5	6.018503e-5	9.027754e-5	6.018503e-5	1.504626e-5
b_n		3.725385e0	-5.226004e0	3.270902e0	-7.705239e-1

These values are coded within the declaration section at the beginning of the program, the listing of the main part of the program is given below.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_adc.c"
#include "stm32f4xx_dac.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_tim.c"
#include "dd.h"
#include "math.h"

short int X1[1024], X2[1024], k = 0;
float Y1[1024], Y2[1024];
#define pi 3.14159

// declare and init IIR weights: 4th order, Chebishew, Low Pass, reference [1]
// +/- 0.5%, -3dB at 0.025 (250Hz) of sampling frequency
float a[5] = {1.504626e-5, 6.018503e-5, 9.027754e-5, 6.018503e-5, 1.504626e-5};
float b[5] = {0          , 3.725385e0, -5.226004e0 , 3.270902e0, -7.705239e-1};

int main ()  {

  GPIOEinit ();
  ADCinit_T5_CC1_IRQ();
  DACinit();
  TIM5init_TimeBase_CC1(8400);           // 8400 == 100us == 10kHz

  while (1) {                            // endless loop
  };
}
```

The program starts with the inclusion of CMSIS files, and proceeds to the declaration section, where the circular buffers are defined. This is followed by the declaration and the initialization of coefficients *a* and *b*. The execution starts with the section “main()”, where the peripherals are configured by four calls of the functions. Next the processor enters the empty infinite loop.

The important stuff again happens in the interrupt function, listed below.

```
void ADC_IRQHandler(void)
{
  GPIOE->BSRRL = BIT_8;
  X1[k] = ADC1->DR;                                                    // 4
  X2[k] = ADC2->DR;                                                    // 5
  float conv = 0;
  for (int m = 0; m<5; m++)                                            // 7
```

```
    conv += a[m] * X1[(k - m) & 1023];                                    // 8
  for (int n = 1; n<5; n++)                                               // 9
    conv += b[n] * Y1[(k - n) & 1023];                                    // 10
  Y1[k] = conv;                                                           // 11
  Y2[k] = (float)X1[k];                                                   // 12
  DAC->DHR12R1 = (int)Y1[k];                                              // 13
  DAC->DHR12R2 = (int)Y2[k];                                              // 14
  k++; k &= 1023;                                                         // 15
  GPIOE->BSRRH = BIT_8;
}
```

Here results from the two ADCs are first stored in the circular buffer in lines 4 and 5. This is followed by the calculation for one input signal only, one convolution for current and past input samples and coefficients a_m, and one convolution for the past results of filtering and coefficients b_n, lines 7 to 10. The complete calculation is performed using floating point arithmetic to assure the required precision, and the result is stored in the output circular buffers in line 11. The unfiltered input signal is copied to the other output buffer for comparison in line 12. Lastly, current values from the output buffers are copied to DACs in lines 13 and 14, and the pointer to circular buffers is updated in line 15.

All calculations and sample management is fenced into two statements to make a bit at port E high at the beginning of calculation and to return the same bit low at the end of calculation. This bit can be used to determine the time needed to execute the interrupt function, which is about 6μs for this example.

[1] Steven W. Smith: The Scientist and Engineer's Guide to Digital Signal Processing

21. PID control

The PID (Proportional Integral Differential) controller is a basic building block in regulation. It can be implemented in many different ways, this example will show you how to code it in a microcontroller and give a simple demonstration of its abilities.

21.1. The theory of PID control

Consider a well stirred pot of water (system), which we need to keep at the desired temperature (reference value, *R*) above the temperature of the surroundings. What we do is we insert a thermometer (sensor) into water and read its temperature (actual value, *X*). If the water is too cold, we turn-on the heater (actuator) placed under the pot. Once the temperature reading on the thermometer reaches the desired value we turn off the heater. The temperature of the water still rises for some time (overshoot), and then starts to decrease. When temperature of the water drops below the desired value we turn-on the heater again. It takes some time before the heater heats-up (this causes an undershoot in temperature) and starts to deliver the heat into the water, but eventually the temperature of the water reaches the desired value again, and the process repeats. What we have is a regulation system, where we act as a controller; we observe the actual value, compare it with the reference value, and stimulate the system based on the result of the comparison, Fig. 21.1.

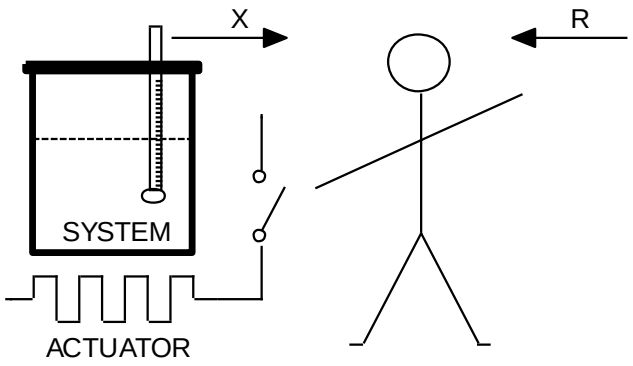


Figure 21.1: A crude example for a regulation

The temperature of the water in the above example never remains at the desired value, but instead wobbles around it. The wobbling depends on the properties of the system, and properties of the sensor and actuator. In order to improve the behavior of the temperature and reduce the wobbling we can improve the regulation process by introducing more complex decisions in the controller. For instance: we could stop the heating some time before the temperature reaches the desired value if we know the amount of overshoot. We could reduce the overshoot also by reducing the amount of heat delivered into the water when the actual temperature becomes close to the desired. There are other possibilities, but they can all be put to life by introduction of a control unit which performs so-called PID regulation.

In terms of regulation theory the above crude system including the actuator and sensor can be described by a second order differential equation, and the regulated system is called a second order. These are best tamed by a PID controller, Figure 22.2.

A PID controller consists first of a unit to calculate the difference (Error) between the desired value Ref and the actual value X. The calculated error signal is fed to three units to calculate the multiple of the error (proportional part, Prop), the rate of changing of the error (differential part, Dif), and the up-to-now sum of the error (integral part, Int). All three components are weighted by corresponding factors (Kp, Kd, Ki) and summed to get the final value (Reg) used by the actuator to influence the system.

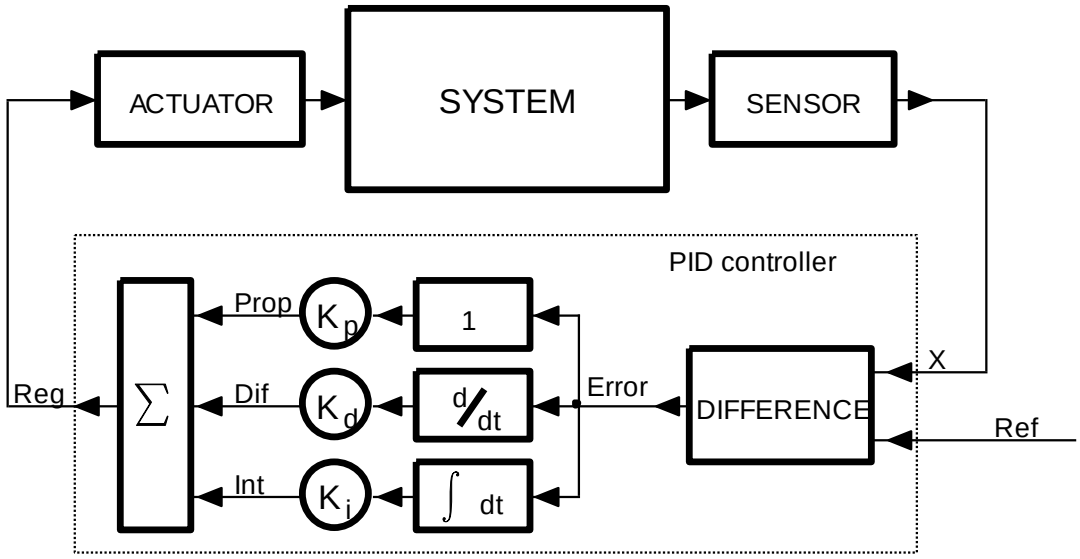


Figure 22.2: A system controlled by a PID controller

21.2. The PID controller implemented in a microprocessor

The microcontroller houses all required building blocks to implement the measurement of the actual and reference value (ADC), the computation of the controller function, and the generation of a signal to influence the system (DAC). When such PID controller is implemented in microcontroller the calculation cannot be continuous as with analog electronic circuits, but must be performed periodically with a period that is short enough compared to the response time of the regulated system. This again calls for periodic sampling, calculation and generation of values. The same programming skeleton as used in FIR and IIR filtering can be re-used for the task of PID control. The initialization of the microcontroller is the same, and all calculation of the controller functions should be performed within the interrupt function. Its listing is given below, the calculation follows Figure 22.2 exactly.

```
void ADC_IRQHandler(void)               // this takes approx 6us of CPU time!                  // 1
{
  GPIOE->ODR |=  0x0100;                // PE08 up                                             // 3
  R[k]  = ADC2->DR;                     // pass ADC -> circular buffer Ref                     // 4
  X[k]  = ADC1->DR;                     // pass ADC -> circular buffer X                       // 5

  Error[k] = R[k] - X[k];               // calculate error                                     // 7
  Prop = Kp * (float)Error[k];          // proportional part                                   // 8
  Dif  = Kd * (float)(Error[k] - Error[(k-1) & 63]) / Ts;  // differential part                // 9
  Int += Ki * (float)Error[k];          // integral part                                       // 10
  Reg[k] = (int)(Prop + Dif + Int);     // summ all three                                      // 11
```

```
  if (Reg[k] > 4095) DAC->DHR12R1  = 4095;    // limit output due to the DAC                  // 13
  else if (Reg[k] < 0) DAC->DHR12R1  = 0;                                                      // 14
       else DAC->DHR12R1  = (int)Reg[k];      // regulator output -> DAC                      // 15
  DAC->DHR12R2  = Error[k] + 2048;            // Error -> DAC                                 // 16
  k = (k + 1) & 63;                     // increment pointer to circular buffer               // 17
  GPIOE->ODR &= ~0x0100;                // PE08 down                                          // 18
}
```

The reference and the actual signals are measured by two ADC, and are stored in corresponding circular buffers *R* and *X* in lines 4 and 5. The calculation of the PID control is performed in lines 7 to 11, and results are sent to DACs in lines 13 to 16. The calculation is complemented by line 17 to increase and bound the pointer to circular buffer size, and by lines 3 and 18 to toggle port E, pin 8. This signal can be used to determine the execution time of the interrupt function.

Lines 7 to 11 are discussed next. First the error signal is calculated in line 7, being the difference between the desired and the actual value. The error signal is stored in its own circular buffer for further use. Next three components *Prop*, *Dif*, and *Int* are calculated. The component *Prop* is just a weighted error value, the weight being *Kp*. The component *Dif* should be a differential of the error value. Here we calculate the differential as a simple difference between the current error and the past error, divided by the time interval and weighted by value *Kd*. The last component is called *Int*, the integral of the error value. This is calculated by simple adding of the weighted current error to the sum of all previous errors; the weight is called *Ki*. Line 11 sums all three components together to form the controller output *Reg*.

Weights *Kp*, *Kd* and *Ki* are set in the main program, and are floating point values. The samples from ADCs are short integer values, and must be converted to floats before the use. The components *Prop*, *Dif* and *Int* are floating point values, and should be declared as such. The sum of all three components is again a floating point value, but should be sent to DAC as a short integer.

The result of PID controller may be outside the range of the DAC; it is best to limit the output from the controller to a value between 4095 and 0, as shown in lines 13 and 14. If the value is already within the range then we can simply send it there as in line 15. The error value is sent to the second DAC for evaluation in line 16; since it may be of either polarity it is best to offset it at the middle of the DAC range.

The listing of the rest of the program is presented next.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_adc.c"
#include "stm32f4xx_dac.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_tim.c"
#include "dd.h"
#include "LCD2x16.c"

#define pi 3.141592653589

int   Ref[64], X[64], k;              // declare circular buffers                            // 12
float Kp = 1.0, Ki = 0.0, Kd = 0.0;   // declare & init params                               // 13
int Error[64];                        // declare error vector                                // 14
int Reg[64];                          // declare past output vector                          // 15
float Prop, Dif, Int = 0;             // declare (& init) vars                               // 16
float Ts = 0.0001;                    // defined by constant 8400 in TIM2->arr;              // 17
```

```
int main ()  {                                                                          // 19
int sw;
  GPIOEinit ();                                                                         // 21
  SWITCHinit();
  ADCinit_T5_CC1_IRQ();
  DACinit();
  TIM5init_TimeBase_CC1(8400);          // 8400 == 100us == 10kHz                      // 25

  LCD_init();                                                                           // 27
  LCD_string("Kp:", 0x00);                                                              // 28
  LCD_string("Kd:", 0x09);                                                              // 29
  LCD_string("Ki:", 0x49);                                                              // 30

  while (1) {                                                                           // 32
    sw = GPIOE->IDR;                                                                    // 33
    if ((sw & S370) && !(sw & S372) && !(sw & S373)) Kp--;   // manually set Kp         // 34
    if ((sw & S371) && !(sw & S372) && !(sw & S373)) Kp++;                              // 35
    if (Kp<0) Kp = 0;     if (Kp > 1000) Kp = 1000;                                     // 36
    if ((sw & S370) &&  (sw & S372) && !(sw & S373)) Kd -= 0.001;  // manually set Kd  // 37
    if ((sw & S371) &&  (sw & S372) && !(sw & S373)) Kd += 0.001;                       // 38
    if (Kd < 0) Kd = 0;   if (Kd > 1) Kd = 1;                                           // 39
    if ((sw & S370) && !(sw & S372) &&  (sw & S373)) Ki -= 0.0001; // manually set Ki  // 40
    if ((sw & S371) && !(sw & S372) &&  (sw & S373)) Ki += 0.0001;                      // 41
    if (Ki < 0) Ki = 0;   if (Ki > 1) Ki = 1;                                           // 42
    LCD_sInt3DG((int)Kp,0x03,1);                          // write Kp                   // 43
    LCD_sInt3DG((int)(Kd*1000),0x0c,1);                   // write Kd                   // 44
    LCD_sInt3DG((int)(Ki*10000),0x4c,1);                  // write Ki                   // 45
    for (int i = 0; i < 5000000; i++)      {};            // waste time                 // 46
  };
}
```

The listing starts with the inclusion of CMSIS files and the declaration of circular buffers in line 12. The weights are declared and initialized next, followed by the declaration of other variables in lines 13 to 17. The line 17 declares and initializes the time interval between successive samples to 100 μs.

The function “main()” starts with calls to the configuration functions for ADC, DAC, timer TIM5 and ports. All functions used were already described in previous chapters. The timer TIM5 is initialized to issue interrupt requests every 100 μs in line 25.

Lines 27 to 30 configure the LCD screen and write introductory text to it.

The quality of PID regulation depends on the values of weights. These can be set by pressing pushbuttons on the BaseBoard. Due to the limited number of pushbuttons a combination of them must be pressed to set the desired value. All this is handled in lines 34 to 42. Pushbutton S370 is used to decrease, and pushbutton S371 to increase the value of the weight selected. The weight to be changed is selected by the combination of pushbuttons S372 and S373:

- If none of them is pressed we are dealing with weight *Kp*,
- If S372 is pressed and S373 is not pressed we are dealing with weight *Kd*,
- If S372 is not pressed and S373 is pressed we are dealing with weight *Ki*.

Line 34 for instance checks the three pushbuttons S370, S372 and S373, and decreases the value of weight *Kp* is correct situation is detected. Line 35 is used to check the state of pushbuttons for increasing of the same weight. Line 36 is used to limit the value of this weight to a reasonable value in case the user requests too much. All three weights are written to the LCD screen in lines 43 to 45.

The program can be tested by adding a simple second order system between the DAC output (Reg, DAC1) and ADC input (actual value, ADC1->DR, ADC_IN1 input). The other ADC (ADC2->DR, ADC_IN2 input) is used to read the reference value Ref. Two serially connected RC circuits are used as a substitute for the second order system; this simplifies the demonstration. Additionally, the desired value Ref can be generated using a function generator, as can the interference signal Intf. The complete connection of the demo circuit is given in Fig. 22.3.

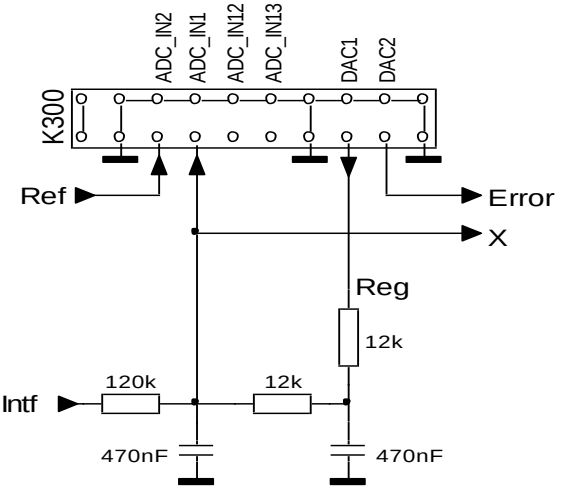


Figure 22.3: The connection for experiment

Figures 22.4 to 22.7 give the actual values (X, red) for the circuit in Fig. 22.3 for different values of K_p, K_d, and K_i. The interference signal Intf is kept at zero, and the desired value (Ref, blue) is a squarewave with the frequency of 10Hz. The offset of the Ref signal is set close to the middle of the ADC scale. The horizontal scale is in seconds, while the vertical scale is in volts.

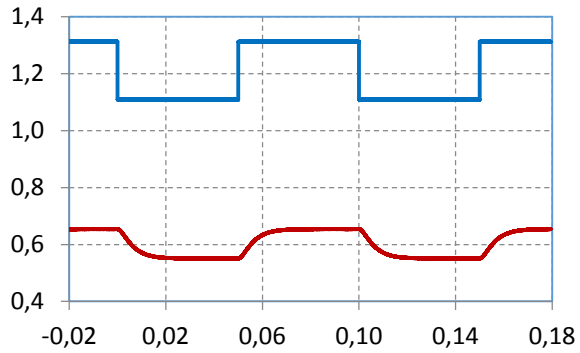


Figure 22.4: Kp = 1, Kd = 0, Ki = 0; Proportinal gain is too low and the actual value (red) does not reach the desired value (blue)

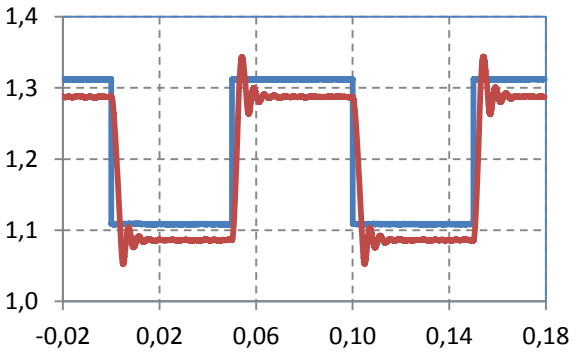


Figure 22.5: Kp = 50, Kd = 0, Ki = 0; Proportional gain is high, the actual value (red) is closer to the desired value (blue), but oscillations become visible

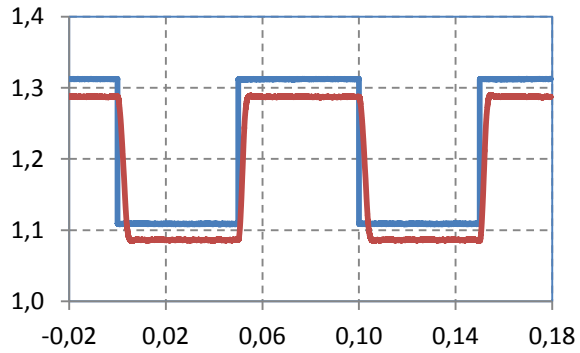


Figure 22.6: Kp = 50, Kd =40, Ki = 0; Differential gain smooth out oscillations, but the actual value (red) is still offseted from the desired value (blue)

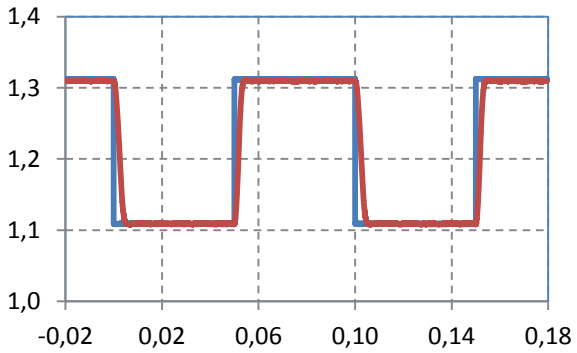


Figure 22.7: Kp = 50, Kd = 40, Ki = 40; The integral part pushes the actual value up to become almost identical to the desired value

The next set of diagrams on Figures 22.8 to 22.11 give responses (red) of the regulated system to the interference signal (blue) while the reference signal (not shown) is kept at a constant value of

1.21V. It is expected that the response is also constant if the regulator does its job properly. Scales are the same as for the previous set of figures.

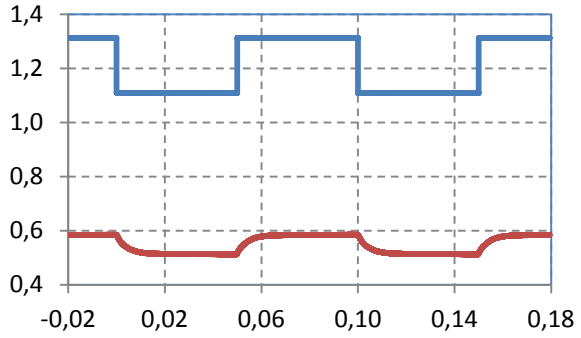


Figure 22.8: Kp = 1, Kd = 0, Ki = 0; Proportional gain is too low and the influence of the interfering signal is significant

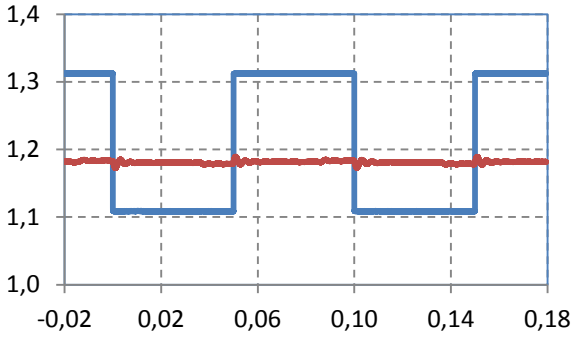


Figure 22.9: Kp = 50, Kd = 0, Ki = 0; Proportional gain is high, the actual value (red) is closer to 1.21V, but oscillations caused by the interfering signal are visible

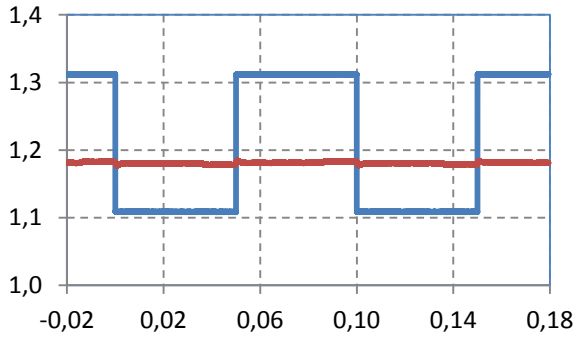


Figure 22.10: Kp = 50, Kd =4 0, Ki = 0; Differential gain smooth out the oscillations, but the actual value (red) is still not the same as the desired value (1.21V)

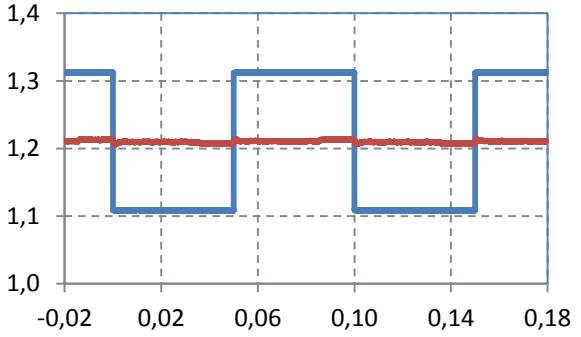


Figure 22.11: Kp = 50, Kd = 40, Ki = 40; The integral gain pushes the average of the actual value up to become the same as the average of the desired value (1.21V)

22. PLL and frequency demodulation

The PLL (Phase Locked Loop) describes the common way for generating a signal with a frequency which is in a fixed relation with the frequency of the reference signal (like multiple). It can also be used for measuring of frequency and the frequency demodulation. The example demonstrates the implementation of such block in software. The digital input signal f_{in} is connected to port E, pin 15, and the microcontroller generates a sine wave signal with the same frequency and phase DAC1 output. The output from DAC2 is used as a frequency demodulated output.

22.1. The theory of PLL

A Phase Locked Loop (PLL) is another commonly used block in digital electronics. The PLL block is capable of generating a signal f_{VCO} with a frequency which is the same as the frequency of the input signal f_{IN}; it is given in Fig. 22.1 in its basic form. By adding two dividers (one in series with each of the input signals to the phase comparator) the block generates a signal f_{VCO} which is the ratio of the two division factors multiplied by the frequency of the input signal f_{IN}. The same block can be used for frequency demodulation of the input signal f_{IN}; the signal T_P is proportional to the frequency of the input signal f_{IN}.

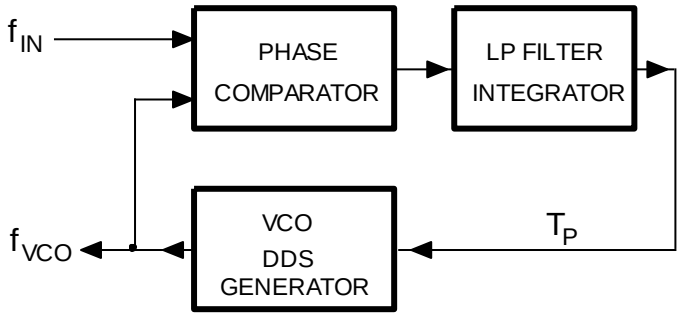


Figure 22.1: Graphical representation of the PLL block

In the following example we will program the PLL into the microcontroller and obtain a frequency meter and frequency demodulator at the same time. For those with experience in digital electronics: there are basically two types of phase comparators, XOR gate and a memory circuit (referred as Type I and Type II in typical PLL chip CD4046); we will implement the memory circuit in this example since it prevents PLL to lock on harmonics of the input signal.

The frequency of the local oscillator (VCO, DDS GENERATOR) f_{VCO} depends on the control signal Tp, and can be the same as the frequency of the input signal f_{IN}, for a certain value of control signal Tp, as shown in Fig. 22.2. In this case the PLL block is locked to the input signal in frequency and phase, and no further action is needed; signal T_p does not need to be adjusted. Arrows indicate moments of sampling of the two signals, and numbers their current value.

When phases (and frequencies) of the two signals differ, two scenarios are possible. The local signal f_{VCO} can be delayed compared to the input signal f_{IN}, as shown in Fig. 22.3, left; in this case the frequency of the local signal f_{VCO} must be increased to align the edges, therefore the control signal T_p

must be increased. Alternatively, the local signal f_{VCO} can come ahead of the input signal f_{IN} as shown in the same figure, right; in this case the frequency of the local signal must be decreased (and T_p as well) to match the edges.

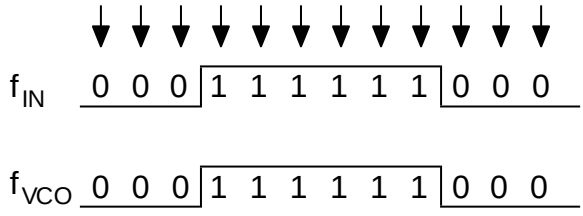


Figure 22.2: PLL block is locked to the input signal;
f_{VCO} is equal to f_{IN} in frequency and phase, no adjustments of T_p are needed

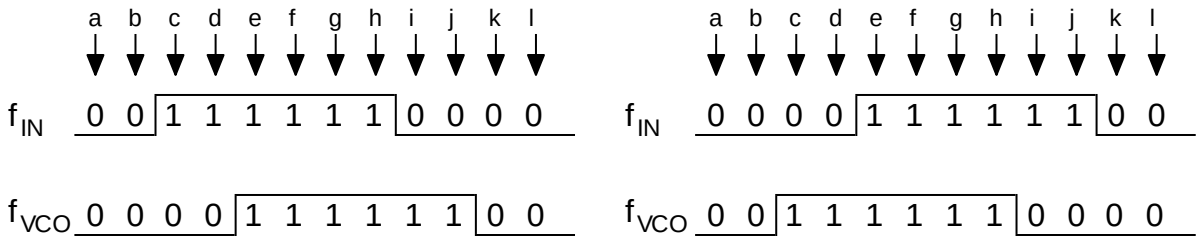


Figure 22.3: Two possible scenarios: left => frequency of signal f_{VCO} must be increased to match the phase;
right => frequency of the signal f_{VCO} must be decreased to match the phase

From Fig. 22.3, left, we can deduce to increase the frequency (therefore control signal T_p) while the input signal f_{IN} is high and the local signal f_{VCO} is low, and also while the input signal f_{IN} is low and the local signal f_{VCO} is high. The same can be deduced for decreasing the frequency from Fig. 22.3, right. What to do, then?

Consider the current and past samples of the input signal. The consecutive samples can be arranged to form an array of bits having value of either zero or one. Such array, when short, can comprise an integer variable; bit 0 (LSB) of the variable belongs to the current sample of the input signal, bit 1 belongs to the previous sample, bit 2 to the pre-previous sample... , like 001111110000 for the Fig. 22.3, left, top. When two digital signals (for f_{IN} and f_{VCO}) are sampled simultaneously, consecutive samples can be arranged in a common integer variable in such a way that odd bits (bit 1, bit 3,…) of the integer variable represent signal f_{IN}, and even bits (bit 0, bit 2,…) signal f_{VCO}. Additionally, let the least significant two bits represent the current values of both signals, and adjacent two more significant bits past values of the same signals, Fig. 22.4. Only four bits are important; for instance, at time **c** (the third sample, Fig. 22.3, left) the corresponding integer number reads 0010_b.

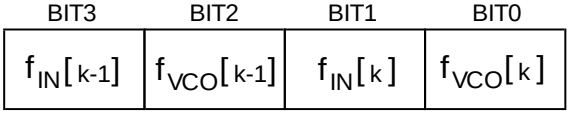


Figure 22.4: The arrangement of bits in variable InPat for the phase detection

Using this construct we can now identify the following situations:

- When signal f_{IN} and f_{VCO} have equal values we should not change the frequency of the VCO since we have no significant information on the relation of two frequencies (control signal T_p). The relation is hidden in the position of the edges of both signals, and we should concentrate on those.

- When signals have different values, we should change frequency of the VCO:
  - The frequency should start increasing when the constructed integer *InPat* becomes 0010b, and keep increasing until both signals become equal. This situation means that the leading edge at f_{IN} came earlier than the leading edge at f_{VCO}, so the frequency of f_{VCO} should be increased (T_p increased).
  - The frequency should also start increasing when the constructed integer *InPat* becomes 1101b, and keep increasing until both signals become equal. This situation means that the falling edge at f_{IN} came earlier than the falling edge at f_{VCO}, so the frequency of f_{VCO} should be increased (T_p increased).
  - The frequency should start decreasing when the constructed integer *InPat* becomes 0001b, and should keep decreasing until both signals become equal. This situation means that the leading edge at f_{VCO} came earlier than the falling edge at f_{IN}, so the frequency of f_{VCO} is too big and should be decreased (T_p decreased).
  - The frequency should also start decreasing when the constructed integer *InPat* becomes 1110b, and should keep decreasing until both signals become equal. This situation means that the falling edge at f_{VCO} came earlier than the falling edge at f_{IN}, so the frequency of f_{VCO} is too high and should be decreased (T_p decreased).

The unit performing this functions (constructing the integer value and calculating the required value of T_p) replaces both the phase comparator and the LP filter / integrator in the block diagram in Fig. 22.1.

22.2. The implementation of the PLL

The program to implement for PLL unit is based on the program for the DDS, chapter 16. The structure of the program is the same: the timer TIM5 is used to trigger periodic interrupts every 10 μs. All PLL processing is performed within the interrupt function for timer TIM5. The main program is used only to display current frequency of the DDS generator which is the same as the frequency of the input signal, when PLL is locked.

The interrupt function is presented in the listing bellow.

```
void TIM5_IRQHandler(void)      // PLL takes approx 600 ns of CPU time!
{
  GPIOE->BSRRL =  BIT_8;        //
  TIM_ClearITPendingBit(TIM5, TIM_IT_Update); // clear interrupt flag                          // 4
  PortE = GPIOE->IDR & BIT_15;  // read input signal                                           // 5
  f_VCO = ptrTable   & 0x8000;  // this is locally generated signal                           // 6
  InPat = (InPat << 2) & 0x0c;  // construct pattern for phase detector                       // 7
  if (PortE) InPat += 2;        //                                                            // 8
  if (f_VCO) InPat += 1;        //                                                            // 9
  if (PortE == f_VCO)       dTp =  0;   // if equal signals (frequencies)                    // 10
  else { if (InPat == 0x02) dTp =  1;   // if frequency too low                              // 11
         if (InPat == 0x0d) dTp =  1;   //                                                  // 12
         if (InPat == 0x01) dTp = -1;   // if frequency too high                            // 13
         if (InPat == 0x0e) dTp = -1;   //                                                  // 14
  };
  K += dK;                              // correct time interval                            // 16
  if (Tp > 0x8000) Tp = 0x8000;         // but not too much                                 // 17
  if (Tp < 0x0080) Tp = 0x0080;         //                                                  // 18
  ptrTable = (ptrTable + Tp + (dTp << 8)) & 0xffff;  // update pointer to table             // 19
  GPIOE->ODR = (GPIOE->ODR & ~BIT_10) | (f_VCO >> 5);// digital out - f_VCO                 // 20
  DAC->DHR12R1 = (Table[ptrTable >> 4]) + 2048;      // analog signal -> DAC                // 21
```

```
  DAC->DHR12R2 = K >> 2;                              // frequency -> DAC                        // 22
  GPIOE->BSRRH = BIT_8;        //
}
```

As already discussed the line 4 is used to clear the flag for the pending interrupt request in timer TIM5. The input signal f_{IN} is connected to port E, pin 15, the most significant bit of the short integer read into the processor by command in line 5; the AND function is used to remove the presence of any other bits and retain the result as *PortE*. Note that the pointer into the table *ptrTable* as used in the example on DDS generation in chapter 16 is also a short integer, and consecutive values of its most significant bit form a square wave signal with a frequency that equals the frequency of the generated sine wave. The digital signal *f_VCO* representing the sine wave from the DDS is constructed in line 6 using the AND function from the MSB of the table pointer.

The integer *InPat* is constructed in lines 7 to 9. Current value of the integer *InPat* is shifted left for two places, and only the least significant four bits are retained. The least significant two bits are then modified following the values of both input signal *PortE* and the locally generated signal *f_VCO.*

Once the variable *InPat* is constructed the software decides what to do with the frequency of the DDS generator in lines 10 to 14. If both signals *f_VCO* and *PortE* are the same the variable *dTp* (delta *Tp*, for this amount the frequency should change) is set to zero. However, when signals are different, all four possibilities from the former bulleted list are checked and the variable *dTp* is set accordingly.

The variable *dTp* is next used to update the control variable *Tp* (line 16) which define the frequency of the signal f_{VCO}. It might happen that the update pushes control variable *Tp* out of the acceptable range, so *Tp* is checked and bound in lines 17 and 18.

The next statement in line 19 updates the pointer *ptrTable* to table with samples of the output signal. The last term in the sum needs to be added to ensure the stability of the PLL loop (check the theory). The last three statements take care of generating the actual signals at pins of the microcontroller. The digital signal f_{VCO} is generated at port E, bit 10, and the analog version is generated at the DAC1. The second DAC2 is used to output the control variable *Tp*, therefore the analog voltage representing the frequency of the locally generated signal f_{VCO}. When a frequency-modulated signal is used ad f_{IN}, the output of the DAC2 is the demodulated version of the input signal.

The listing of the rest of the program is given below.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_tim.c"
#include "stm32f4xx_dac.c"
#include "LCD2x16.c"
#include "math.h"
#include "dd.h"

int Table[4096], ptrTable, Tp = 655, dTp = 0;
char InPat = 0;
int PortE, f_VCO;

int main ()  {

  // Table init for analog output
  for (ptrTable = 0; ptrTable <= 4095; ptrTable++)
    Table[ptrTable] = (int)(1850.0 * sin((float)ptrTable / 2048.0 * 3.141592653));

  LCD_init(); LCD_string("Fin=", 0x01); LCD_string("Hz", 13);
  GPIOEinit_8to11();
```

```
  DACinit();
  TIM5init_TimeBase_ReloadIRQ(840);    // 840 == 10us == 100kHz

  while (1) {
    LCD_uInt16((int)(Tp * 100000 / 65536),0x07,1);   // display frequency
    for (int i = 0; i < 5000000; i++) {};            // waste time
  };
}
```

This listing starts with the inclusion of CMSIS files and the declaration of the variables used. The function “main()” starts with the initialization of the table to generate analog signal from as already explained in chapter 16, and continues to the configuration functions for LCD, port E, DAC and timer TIM5. Note the function for the configuration of port E, which is the modified version of the function given in chapter 4. Only pins 8, 9, 10, and 11 are configured as outputs, other pins are inputs. The endless while loop is used only to output the current frequency of the DDS generator to the LCD display.

23. Lock-in detection

Lock-in detection is a technique for the detection of the amplitude and phase of a sinusoidal signal with a known frequency. Its distinctive advantage over other detection techniques is that it can selectively determine the amplitude and phase of a signal at just the desired frequency (within a narrow band), and reject signals with other frequencies. This significantly boosts the signal to noise ratio of the detection system. The example demonstrates the implementation of a lock-in detection in software.

The microcontroller first generates a sinusoidal signal at the DAC output using the DDS technique described in Chapter 16. This signal drives the externally connected RC network that alters its amplitude and phase. The resulting signal is measured by the ADC within the microcontroller. The software calculates the amplitude of the measured signal and its phase using the lock-in detection algorithm in real time.

23.1. The theory of lock-in detection

The block diagram of the lock-in detection algorithm is depicted in Fig. 23.1. The tested system is driven by a sinusoidal signal *REF*, having a predefined amplitude and frequency. The output signal from the tested system is fed to an ADC input, where it gets sampled into signal *X*. Next the signal *X* gets multiplied, once by in-phase version (R0) and once by 90-degree out-of-phase version (R90) of the excitation signal.

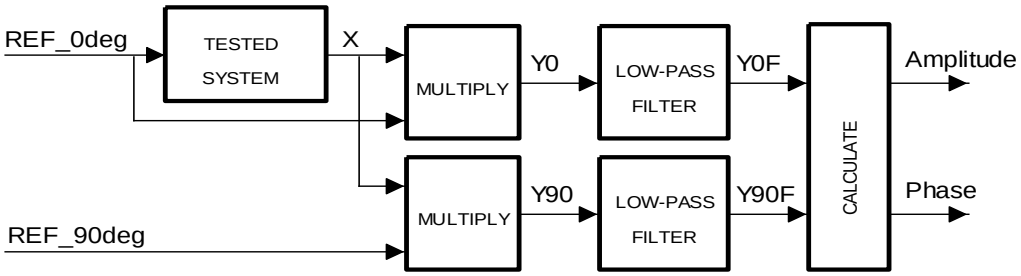


Figure 23.1: The lock-in detection: block diagram

Let the excitation signal *REF* be:

$$REF = R \sin \omega t$$

Then the input signal *X* is:

$$X = A \sin(\omega t + \varphi)$$

The input signal *X* has the same frequency as the excitation signal *REF*, but may differ in amplitude and phase. The multiplication of the input signal *X* with the two versions of the excitation signal gives the intermediate signals Y_0 and Y_{90}:

$$Y_0 = R \sin \omega t \ \cdot \ A_1 \sin(\omega t + \varphi) = \frac{RA_1}{2} \left[\cos(\varphi) - \cos(2\omega t + \varphi)\right]$$

$$Y_{90} = R\cos\omega t \ \cdot \ A\sin(\omega t + \varphi) = \frac{RA}{2}\left[\sin(\varphi) + \sin(2\omega t + \varphi)\right]$$

Each of these intermediate signals is composed of two parts. One part is constant in time, the other oscillates with double the frequency of the excitation signal. Signals Y_0 and Y_{90} are sent through two low-pass filters to get rid of the AC components, and get signals Y_{0F} and Y_{90F}; we want to set the corner frequency of the low pass filter as low as practically possible to reject as much of AC component as possible.

$$Y_{0F} = \frac{RA}{2}\cos\varphi \qquad\qquad Y_{90F} = \frac{RA}{2}\sin\varphi$$

These two signals contain the information on the amplitude *A* of the signal *X* and its phase φ, we only need to calculate them using simple mathematics:

$$\sqrt{Y_{0F}^2 + Y_{90F}^2} = \frac{R\cdot A}{2} \qquad \rightarrow \qquad A = \frac{2}{R}\sqrt{Y_{0F}^2 + Y_{90F}^2}$$

$$\frac{Y_{90F}}{Y_{0F}} = \frac{\sin\varphi}{\cos\varphi} \qquad \rightarrow \qquad \varphi = \tan^{-1}\frac{Y_{90F}}{Y_{0F}}$$

Calculating the geometrical sum of signals Y_{0F} and Y_{90F} gives a weighted amplitude of the detected signal, and calculation of the arc tangent of the quotient of the two signals gives the phase angle.

The advantage of the lock-in detection can be clearly seen when the input signal *X* is a sum of two components: the first one is the same as above, and the second is an unwanted signal with a frequency ω_Z ($\omega_Z \neq \omega$) and amplitude *Z*.

$$X = A\sin(\omega t + \varphi) + \ Z\sin(\omega_Z t + \varphi_Z)$$

Since the system is linear the processing of the first component remains as it was described above. The processing of the second component is analyzed below. The second component is multiplied with the excitation signal and its 90 degree sibling to get components Y_{0Z} and Y_{90Z}.

$$Y_{0Z} = R\sin\omega t \ \cdot Z\sin(\omega_Z t + \varphi_Z) = \frac{RZ}{2}\left[\cos\big((\omega - \omega_Z)t - \varphi_X\big) - \cos\big((\omega + \omega_Z)t + \varphi_X\big)\right]$$

$$Y_{90Z} = R\cos\omega t \ \cdot Z\sin(\omega_Z t + \varphi_Z) = \frac{RZ}{2}\left[\sin\big((\omega + \omega_Z)t + \varphi_X\big) - \sin\big((\omega - \omega_Z)t - \varphi_X\big)\right]$$

All components in both expressions are harmonic signals with frequency other than zero. The low-pass filter being the next stage in signal processing rejects AC components, and they do not affect the outputs of filters Y_{0F} and Y_{90F}. The outputs of both filters depend only on the input signal which has the same frequency as the excitation signal.

In practice, low pass filters have a finite corner frequency f_c and finite attenuation steepness in the transition region between pass-band and stop-band, and the lock-in detection system is then sensitive for signals having exactly the frequency of the excitation signal f_{REF} and signals having a frequencies close to the frequency of the excitation signal. However, the corner frequency f_c can be made small and the steepness can be increased to reject more interfering components at the expense of filter complexity and a slower response to the change of the signal at the excitation frequency. The Fig. 23.2 gives the frequency spectrum of the sensitivity for such a system.

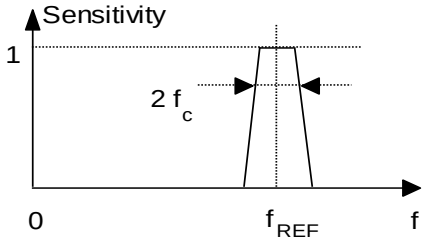


Figure 23.2: The relative sensitivity versus frequency of a lock-in detection system

23.2. The implementation of the lock-in detection

The lock-in detection will be tested on a simple passive electronic circuit that can change the amplitude and phase of a harmonic signal. The schematic diagram is given in Fig. 23.3. The microcontroller generates the driving signal *REF_0deg* at its analog output DAC1. The driving signal should have a frequency of about 1 kHz in this experiment since the values of resistor and capacitors are adjusted to this frequency to give an adequate amplitude and phase change. The potentiometer P1 is used to adjust the output amplitude, and the potentiometer P2 is used to adjust the output phase. The adjustment of phase and amplitude is not totally independent; changing the amplitude modestly affects the phase setting and vice versa. The output *X* of the circuit is applied to the analog input of the microcontroller, ADC_IN2, where it gets sampled and then processed.

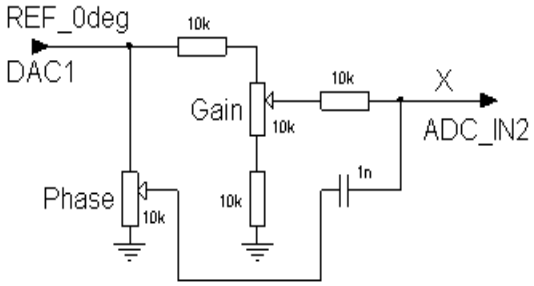


Figure 23.3: The external circuit to the BaseBoard for the lock-in detection experiment

The program is composed of two parts: one is the generation of a sinusoidal signal, and the other is the sampling and data processing of the acquired signals. The program is implemented as an interrupt routine, which is driven by timer TIM5, the time interval between successive interrupts is fixed to 20 μs.

The sinusoidal signal REF is best generated using a DDS technique, described in chapter 16.

The sampling of the input signal is implemented the same way as described in chapters 12 and 13. The timer overflow triggers the ADC to start the conversion, and the end-of-conversion signal from the ADC is used to interrupt the microcontroller and start the interrupt routine ADC_IRQhandler.

The theory behind the lock-in detection data processing is described above. The theory necessitates the calculation of two rather complex functions, namely the square root and the arc tangent. Both require processor time, in this case roughly 15 us. Since this seemed too much the calculation of both functions has been moved from the interrupt routine into the main program. Additionally, since the result of the calculation is shown to the user on an LCD screen, the software does not need to update the display and the calculation very often; this offers additional opportunity to average several consecutive results of measurement and thus improve the stability of the result displayed. The interrupt routine accumulates 64 results, and passes the accumulated version to the main program only on every 64th execution, this will be explained shortly.

The ADC interrupt function for the lock-in detection is presented in the listing below.

```
void ADC_IRQHandler(void)      {
  GPIOE->BSRRL = BIT_8;              // signal start, execution time is about 4us!            // 2

  // DDS generator
  ptrTable = (ptrTable + 655) & 0xffff;         // 655 ==> 1 kHz                            // 5
  DAC->DHR12R1 = Table[ptrTable >> 4] + 2048;   // next excitation                          // 6

  // Measurement
  X[k] = ADC1->DR;     X[k] = ADC2->DR;                                                     // 9

  // multiply with sin and cos components
```

```
  Y0[k]  = (float)X[k] * Table[((ptrTable >> 4) +    0)       ];                                  // 12
  Y90[k] = (float)X[k] * Table[((ptrTable >> 4) + 1024) & 4095];                                  // 13

  // LP, 1st order filter
  Y0F[k]  = 0.99999 * Y0F [(k - 1) & 63] + 0.00001 * Y0 [k];                                      // 16
  Y90F[k] = 0.99999 * Y90F[(k - 1) & 63] + 0.00001 * Y90[k];                                      // 17

  // some averaging to get smoother result
  R_im_acc += Y90F[k];          R_re_acc += Y0F[k];                                               // 20

  // report result every 64th time
  if (k == 0) {                                                                                   // 23
    R_im_fin = R_im_acc;        R_re_fin = R_re_acc;                                              // 24
    R_im_acc = 0;               R_re_acc = 0;                                                     // 25
  };                                                                                              // 26

  k++; k &= 63;                                                                                   // 28
  GPIOE->BSRRH = BIT_8;                 // signal end                                             // 29
}
```

The function commences with setting pin 8, port E, high to signal the start of execution, line 2; the same pin is reset when the function ends, line 29. The generation of the excitation signal is given in lines 5 and 6. First the pointer to the table with sine samples is updated; the incrementing value of 655 gives the frequency of the excitation signal of 1 kHz. Next the entry from the sine table is read and sent to the DAC. The table has 4096 entries, and must be prepared in advance during the initialization process of the microcontroller.

Next the result is read from the ADC in line 9. Since we are using the same initialization routine for ADC throughout this text, and this routine initializes two ADCs, both results should be read one after another to clear their respective interrupt flags. The last result read representing the measured analog signal is stored into the circular buffer named *X* at position *k*. The position pointer *k* increments on every execution of the interrupt routine, line 28. This circular buffer has 64 elements; any number of elements above 2 would do, and 64 is selected arbitrarily.

As the lock-in detection requires the measured value is next multiplied by the sine and cosine components of the excitation signal. This is simple. The sine component is the same as used for the DDS generator, line 12, and the cosine component can be obtained from the same table using and offset of a quarter of the table length, namely 1024, line 13. Function “& 4095” must be used in line 13 to keep the pointer within the table boundaries. Also note that the calculation is performed by floating point numbers to increase precision; one of the arguments in multiplication is converted to “float” before the multiplication. The results are stored in two circular buffers named *Y0* and *Y90* at location *k*.

Low-pass filtering comes next. One might be tempted to implement the filter described in Chapter 20, the IIR filtering. However, in this case such filter might introduce additional problems. We want a low-pass filter with a very low corner frequency f_c, and such filters can be unstable due to the limited precision of calculations and due to the digitalization of filter coefficients, see the reference [1] for details. To elude such problems a simpler first order low-pass filter can be used. First order low-pass filter has inferior frequency characteristics, but is stable and can have very low corner frequency. Its difference equation is:

$$y_k = \alpha \cdot y_{k-1} + (1 - \alpha) \cdot x_k$$

Factor α determines the corner frequency with an approximate formula:

$$f_c = -\frac{f_s \cdot \ln \alpha}{2\pi}$$

In our case the sampling frequency f_s equals 10^5 Hz, and the factor α equals 0.99999; this gives the corner frequency of about 0.16 Hz. Filters for both signals Y0 and Y90 are implemented in lines 16 and 17 providing signals *FY0* and *FY90*; results reside in two circular buffers.

Next comes the averaging, this is implemented in lines 20 to 26 including. There are two global variables *R_im_fin* and *R_re_fin* declared, and on every 64th execution (line 23) of the interrupt routine the content of these two variables gets updated in line 24. There are two variables *R_im_acc* and *R_re_acc*, these are used to accumulate the results in line 20, and when the accumulated values are passed to the main the accumulating variables are reset in line 25.

The listing of the rest of the program is given below.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_adc.c"
#include "stm32f4xx_dac.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_tim.c"
#include "dd.h"
#include "LCD2x16.c"
#include "math.h"
#define pi 3.14159

short int X[64], k = 0;                                                          // 11
float Y0[64], Y90[64], float Y0F[64], Y90F[64], float Y1[64], Y2[64], Y3[64];    // 12
int Table[4096], ptrTable;                                                       // 13

int main ()  {                                                                   // 15
int del;
  for (ptrTable = 0; ptrTable <= 4095; ptrTable++)                               // 17
    Table[ptrTable] = (int)(1850.0 * sin((float)ptrTable / 2048.0 * 3.14159265));// 18

  LCD_init ();                                                                   // 20
  LCD_string("Pha=          dg", 0x00);                                          // 21
  LCD_string("Amp=          mV", 0x40);                                          // 22

  GPIOEinit ();                                                                  // 24
  ADCinit_T5_CC1_IRQ();                                                          // 25
  DACinit();                                                                     // 26
  TIM5init_TimeBase_CC1(840);           // 840 == 10 us == 100 kHz               // 27

  while (1) {                            // endless loop                        // 29
    for (del = 0; del < 10000000; del++)  {};                                    // 30
    float Phi_fin = (atan(R_im_fin/R_re_fin)) * 180 / pi;                        // 31
    if (Phi_fin < 0)    {LCD_string("-", 0x06);    Phi_fin = -Phi_fin; }         // 32
    else                {LCD_string(" ", 0x06);                        };        // 33
    short Phi_int = (int)Phi_fin;                                                // 34
    short Phi_frc = (int)((Phi_fin - Phi_int) * 10);                             // 35
    LCD_sInt3DG((int)Phi_int , 0x07, 1);                                         // 36
    LCD_char('.');   LCD_char(Phi_frc + '0');                                    // 37
    float Amp_fin = (sqrt(R_im_fin * R_im_fin + R_re_fin * R_re_fin))/128*.838;  // 38
    LCD_sInt16((int)(Amp_fin/256), 0x47, 1);                                     // 39
  };                                                                             // 40
}
```

This listing starts with the inclusion of CMSIS files and the declaration of the variables used. The function “main()” starts in line 17 with the initialization of the table to generate analog signal from as already explained in chapter 16, and continues to the configuration functions for LCD, port E, ADC, DAC and timer TIM5; all configuration functions were already described in previous chapters.

The endless while loop, lines 29 to 40, is used to calculate the amplitude and the phase angle of the signal *X* from the results passed by the ADC interrupt routine, and to display the calculated value at the LCD display.

The endless loop starts with a delay in line 30; we do not need to display results very often, human eyes cannot follow fast changing numbers anyway. The code in line 31 is used to calculate the phase angle *Phi_fin* in degrees following the formula given in the theory.

Lines 32 to 37 contain some wizardry to display a floating point number. The “LCD2x16.c” does not comprise a function for displaying a floating point number, and the “sprintf” function is not used in order to keep the final code short.

- The sign of the floating point number is dealt with first. If the number is negative, then it is made positive in line 32, and the negation sign is shown the the LCD, else the negation sign in cleared in line 33.
- The floating point number is split in integer (*Phi_int*) and fractional (*Phi_frc*) part, fractional part having only one digit, lines 34 and 35.
- The integer part is displayed in line 36, the position at the LCD is selected just after the minus sign, if it exists.
- The decimal point is displayed immediately following the integer part in line 37, followed by the fractional digit.

The amplitude of the signal X is calculated in line 38 by calculating the geometrical sum of the two components. The outcome of the square root calculation is too big for a factor of 8 due to the averaging and still some due to the amplitude of the excitation signal; both is adjusted in the same line by a simple multiplication. The amplitude is then displayed at the LCD screen using a function call in line 39.

24. Phase measurement

When one is interested in the phase relation between two signals REF and X, and these signals are not generated by the measurement system, one can use a derivative of the lock-in detection technique already described in chapter 23. In previous chapter the microcontroller generated the signal REF and is was simple to generate also the quadrature version of this signal (the 90-degree shifted version). Now, such signal must be derived from the REF signal using the Hilbert transform. The rest of the system remains the same.

24.1. The theory

The block diagram of the system to determine the phase φ between two harmonic signals is given in Fig. 24.1. Two harmonic signals *XR* and *XP* are available and the task of the measurement system is to determine the phase between them; both signals have the same frequency ω and adequate amplitudes A_R and A_P to be sampled by the ADC in the microcontroller.

$$X_R = A_R \sin \omega t \qquad X_P = A_P \sin(\omega t + \varphi)$$

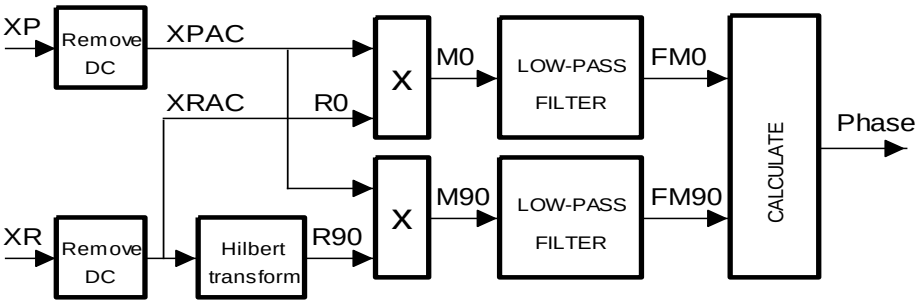


Figure 24.1: The phase measurement: block diagram

The block diagram introduces three new blocks compared to the diagram given in chapter 23, these are the "Remove DC" and the "Hilbert transform" blocks.

The ADC samples the input signals, and the result is between 0 and 4095 for 12 bit ADC; it is always a positive number. The sampled signal therefore rides on a steady DC component of roughly 2048, depending on the DC offset of the ADC. In this example we will strip the DC component off from the sampled signal by a simple DC-removal filter (the "Remove DC" blocks) to demonstrate the process. This operation is not mandatory or essential for the determination of the phase angle, we will simply introduce it here as it looks like a reasonable thing to do in many data processing applications.

Intuitively, the differential equation to strip the DC component off can be written simply as:

$$y_k = x_k - x_{k-1}$$

The y_k is the current output sample, and the x_k and x_{k-1} are the current input and previous input samples. The AC component is proportional to the change in the input value; with the fixed time

interval between samples the above equation returns an approximation of the derivative of the input signal which, of course, is the weighted AC component.

The same difference equation can be derived using knowledge about the z-transform and the influence of poles and zeroes in the z-plane. If we place a zero at the position where $\omega = 0$ in the z-plane, then the system implementing the corresponding transfer function will not let the DC component pass. The position of the zero is illustrated in Fig. 24.2. From here we can write the transfer function *H(z)* and the corresponding difference equation as (note a pole is added in the center of the z-plane; this does not affect the frequency characteristics):

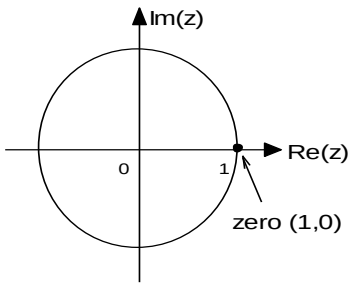
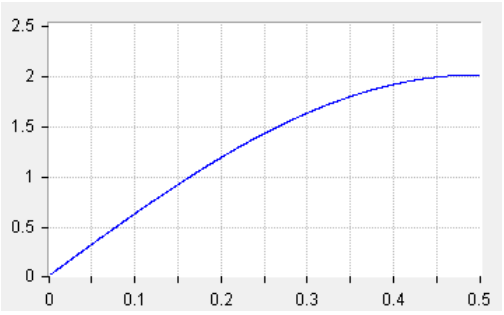


Figure 24.2: The position of a zero in z-plane for a simple DC-removal filter

$$H(z) = \frac{z-1}{z} = \frac{1-z^{-1}}{1} \qquad \rightarrow \qquad y_k = x_k - x_{k-1}$$

The output of this DC-removal filter depends strongly on the frequency of the incoming signal, the amplitude and phase plots are given in Fig. 24.3, and can be calculated using:

$$AMPLITUDE = \left| H(z)|_{z=e^{i\omega t}} \right| \qquad \text{and} \qquad PHASE = \tan^{-1} \frac{IM(H(z))}{RE(H(z))} \Bigg|_{z=e^{i\omega t}}$$



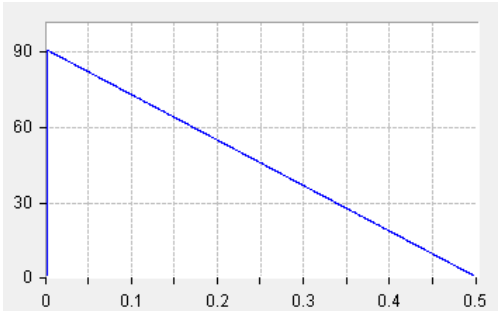


Figure 24.3: Amplitude and phase characteristics of a simple DC-removal filter; horizontal: normalized frequency f/fs, vertical: gain / phase (degrees)

In our experiment the phase characteristic of the filter has no influence on the measurement since equal filters are inserted in both X_R and X_P signals and affect both equally. The reduction of the amplitude at low frequencies might be more of a problem, and can be avoided by modifying a pole-zero diagram. When a pole is inserted close to a zero but inside the unity circle (Fig. 24.4) at a distance *r* from the center the transfer function and the difference equation change to:

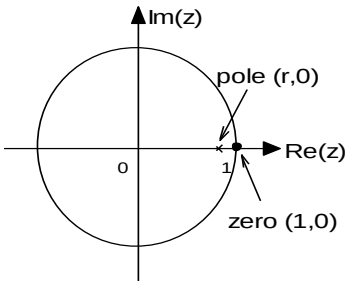


Figure 24.4: The position of a zero and a pole in z-plane for a better DC-removal filter

$$H(z) = \frac{z-1}{z-r} = \frac{1-z^{-1}}{1-rz^{-1}} \qquad \rightarrow \qquad y_k = ry_{k-1} + x_k - x_{k-1}$$

This changes the amplitude and phase plots to the ones given in Fig. 24.5. Note the gain characteristics which is mostly flat now. Both versions of difference equations are simple to implement. In this case the first, the simpler version of differential equation, suffices and will be implemented in software.

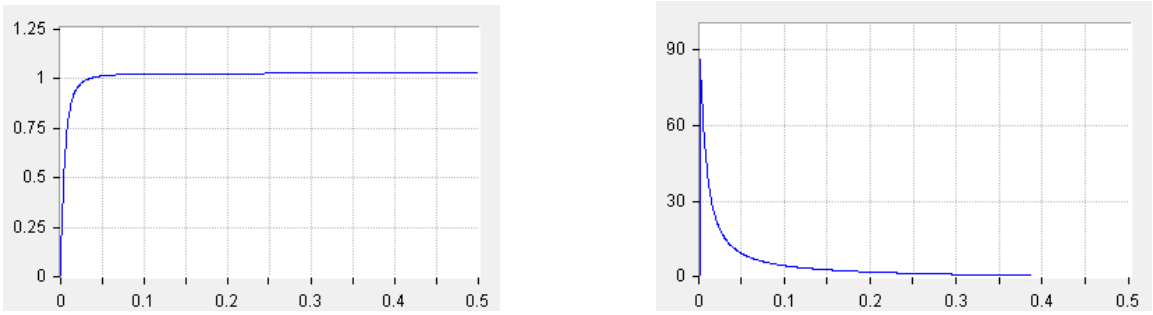


Figure 24.5: Amplitude and phase characteristics of a better DC-removal filter; horizontal: normalized frequency f/f_s, vertical: gain / phase (degrees)

The implementation of the Hilbert transform in a FIR filter was discussed in chapter 19. It was stated that the coefficients of the filter core for a Hilbert transform are given as:

$$h_m = \frac{-1 + (-1)^m}{\pi m}$$

Due to the limited time available between two neighboring samples to calculate the convolution for FIR filter, the number of coefficients in filter core is always restricted, and this affects the frequency characteristics of such a filter; the characteristics can be determined by calculating a Fourier transform of the filter core, and the result for $|m| \leq 32$ is given in Fig. 24.6, blue line. The characteristics is not ideal:

- the amplitude of the filter response changes a lot with the frequency, and
- the amplitude of the response approaches zero at DC and at 0.5 of the sampling frequency.

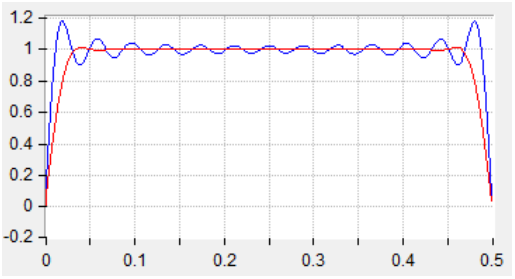


Figure 24.6: Amplitude characteristics for a Hilbert transform with m=32; vertical: normalized amplitude, horizontal: normalized frequency f/f_s

The first obstacle can be significantly reduced by implementing a windowing function, as already mentioned in chapter 19; the effect of windowing is shown by the red line in the same figure.

The second obstacle can be reduced by increasing the number of coefficients in the filter core (you do not want to do this since this increases the time to calculate the convolution) or by carefully selecting the sampling frequency. Suppose the sampling frequency is four times the frequency of the input signals, then we are working at $0.25 \cdot f_s$, and the amplitude characteristics (the red one, windowed coefficients) from Fig. 24.6 is flat in the region. Actually, there is a broad range of frequencies we can use to sample the incoming signal; we must only stay within the flat region of the amplitude characteristics (red) from Fig. 24.6. We could even reduce the number of coefficients in the filter core for the Hilbert transform, get flat response of the filter in smaller central region, and select the sampling frequency to work within this flat region.

It would not be wise to select the sampling frequency that is exactly four times the frequency of the input signals though. The result of multiplication of two signals with the same frequency is composed of two components: one is a DC signal, the other is a signal with a double frequency; $f_s/2$ in this case. According to the Nyquist criteria a signal with frequency of $f_s/2$ behaves like a DC signal, and this comes straight through the low-pass filter! This is not what we want! The same is true also for input signals with frequencies between 0.25 and 0.30 times the sampling frequency; the product contains a component that comes through the low-pass filter and corrupts the measurement.

We have learned that the sampling frequency should be matched to the frequency of the input signal; not precisely, a wide range of sampling frequencies is acceptable for this experiment, but there are also combinations to be avoided.

The rest of the theory was already described in chapter 23 and will not be repeated here.

24.2. The implementation of the lock-in detection

The phase measurement as described above will be tested using an external sine-wave generator and a simple RC passive network to shift the sine-wave. The schematic diagram is given in Fig. 24.7. The signal from the function generator, being a sine-wave, is connected to the network and also to the

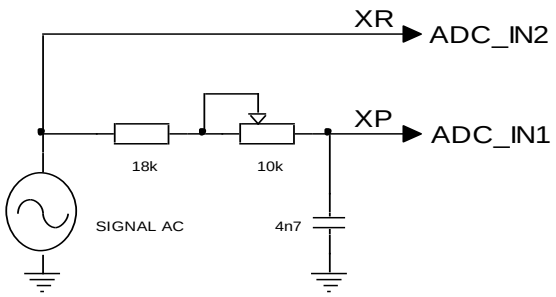


Figure 24.7: The external circuit to the BaseBoard for the phase measurement experiment

ADC input ADC_IN2 at the microcontroller. The values of components in the network are calculated to give a phase shift of about 45 degrees for sinusoidal signals with a frequency of 1500 Hz. The phase shift can be altered by the rotation of the potentiometer.

The program is a modified version of the program from the previous chapter. It is implemented as an interrupt routine, which is driven by timer TIM5, the time interval between successive interrupts is fixed to 100 μs, giving the sampling frequency which is about 6 times the frequency of the input signal.

The sampling of the input signal is implemented the same way as described in chapters 12 and 13. The timer overflow triggers the ADC to start the conversion, and the end-of-conversion signal from the ADC is used to interrupt the microcontroller and start the interrupt routine ADC_IRQhandler.

As in the previous chapter the calculation of a complex function (the arc tangent) is not implemented within the interrupt routine. Instead, the consecutive results are accumulated and then the sum of 128 consecutive results is passed to the main program, where the calculation is performed.

The filter implemented could be the same as used in the previous chapter, but here the IIR version is used, as described in chapter 20. The same filter coefficients are used.

The ADC interrupt function for the lock-in detection is presented in the listing below.

```
void ADC_IRQHandler(void)       {
  GPIOE->BSRRL = BIT_8;                  // signal start, execution time is about 6 us !       // 2

  XR[k] = ADC1->DR;                      // to clear ADC IRQ flag                              // 4
  XP[k] = (ADC2->DR & 0xfff);            // acquire                                            // 5
  XRAC[k] = XR[k] - XR[(k-1) & 127];    // remove DC component                                // 6
  XPAC[k] = XP[k] - XP[(k-1) & 127];    // remove DC component                                // 7

  // derive cos component using Hilbert transform
  float conv = (float)XRAC[(k - 32) & 127] * w[0];                                           // 10
  for (short m = 1; m < 32; m += 2)                                                          // 11
    conv += w[m] *(XRAC[(k - 32 + m) & 127] - XRAC[(k - 32 - m) & 127]);                     // 12
  R0[k] = (float)XRAC[(k - 32) & 127];                                                       // 13
  R90[k] = conv;                                                                             // 14
```

```
  M0[k]  = (float)XPAC[(k - 32) & 127] * R0[k];    // multiply with sin                    // 16
  M90[k] = (float)XPAC[(k - 32) & 127] * R90[k];   // multiply with cos                    // 17

  // IIR low-pass filter for sine component
  conv = 0;                                                                               // 20
  for (short m = 0; m<5; m++)   conv += a[m] * M0[(k - m) & 127];                         // 21
  for (short n = 1; n<5; n++)   conv += b[n] * FM0[(k - n) & 127];                        // 22
  FM0[k] = conv;       FM0acc += conv;                                                    // 23

  // IIR low-pass filter for cosine component
  conv = 0;                                                                               // 26
  for (short m = 0; m<5; m++)   conv += a[m] * M90[(k - m) & 127];                        // 27
  for (short n = 1; n<5; n++)   conv += b[n] * FM90[(k - n) & 127];                       // 28
  FM90[k] = conv;      FM90acc += conv;                                                   // 29

  // average and report result every 128-th time
  if (k == 0)   {                                                                         // 32
    FM0ave  = FM0acc;  FM0acc  = 0;                                                       // 33
    FM90ave = FM90acc; FM90acc = 0;                                                       // 34
    PhaseDisplay = 1;                                                                     // 35
  };                                                                                      // 36

  k++; k &= 127;                                                                          // 38
  GPIOE->BSRRH = BIT_8;               // signal end                                       // 39
}
```

The function commences with setting pin 8, port E, high to signal the start of execution, line 2; the same pin is reset when the function ends, line 39.

Next the result is read from the two ADCs in lines 4 and 5. This returns signals *XR* and *XP*, which are placed into two circular buffers at position *k*. The position pointer *k* increments on every execution of the interrupt routine, line 38. This circular buffer has 128 elements.

The AC-removal filter is implemented in lines 6 and 7 for each signal separately. A simpler version of the filter is used, and results are stored in two circular buffers *XRAC* and *XPAC*.

Next the 90-degree delayed signal is derived out of the signal *XRAC*. The derivation is based on the Hilbert transform filter. The convolution is calculated in lines 10 to 12, and the original version of the signal *XRAC* and its 90 degree delayed version are then stored in circular buffers named R0 and R90 in lines 13 and 14.

The multiplication of the phase shifted signal *XPAC* with the reference is given in lines 16 and 17 to obtain signals *M0* and *M90*. These are then filtered using the IIR version of the filter in lines 20 to 23 and 26 to 29. Finally, lines 32 to 36 implement the passing of the accumulated results to the main part of the program.

The listing of the rest of the program is given below.

```
short int XR[128], XP[128], XRAC[128], XPAC[128], k = 0;
float w[32];
float R0[128], R90[128], M0[128], M90[128], FM0[128], FM90[128];
float FM0ave, FM90ave, FM0acc, FM90acc;
short PhaseDisplay = 0;

// declare and init IIR weights: 4th order, Chebishew, Low Pass, reference [1]
// +/- 0.5%, -3dB at 0.025 (250Hz) of sampling frequency (10kHz)
float a[5] = {1.504626e-5, 6.018503e-5, 9.027754e-5, 6.018503e-5, 1.504626e-5};
float b[5] = {0          , 3.725385e0, -5.226004e0 , 3.270902e0, -7.705239e-1};
```

```
int main ()  {                                                                          // 12

  // calculate Hilbert core for FIR filter
  w[0] = 0;                                    // central weight: 2 x fc / fs         // 15
  for (short m = 1; m < 32; m++)                                                       // 16
    w[m] = (-1.0 + cos(pi * m)) / (pi * m);    // other weights                       // 17
  for (short m = 1; m < 32; m++)                                                       // 18
    w[m] = w[m] * cos(pi / 2 * m / 31.0);      // windowing                           // 19

  LCD_init ();         LCD_string("Phase=", 0x40);                                     // 21

  GPIOEinit ();        ADCinit_T5_CC1_IRQ();          DACinit();                       // 23
  TIM5init_TimeBase_CC1(8400);          // 8400 == 100us == 10kHz                     // 24

  while (1) {                           // endless loop                              // 26
    if (PhaseDisplay) {                                                               // 27
      PhaseDisplay = 0;                                                               // 28
      float Phase = (atan (FM90ave / FM0ave)) / 3.14159 * 180;                        // 29
      if (Phase < 0)    {LCD_string("-", 0x47);    Phase = -Phase; }                  // 30
      else              {LCD_string(" ", 0x47);                    };                 // 31
      short PhaseInt = (int)(Phase);                                                  // 32
      short PhaseFrc = (int)((Phase - PhaseInt) * 10);                                // 33
      LCD_sInt3DG((int)PhaseInt, 0x48, 1);                                            // 34
      LCD_char('.');    LCD_char(PhaseFrc + '0'); LCD_char(0xdf);                     // 35
      for (int j = 0; j<2000000; j++){};      // waste some time                      // 36
    };                                                                                // 37
  };                                                                                  // 38
}                                                                                     // 39
```

This listing starts with the inclusion of CMSIS files (not shown) and the declaration of the variables used. The function “main()” starts in line 12 with the initialization of the filter coefficients for the Hilbert transform; the initialization is copied from Chapter 19, the FIR filtering. The program continues to the configuration functions for LCD, port E, ADC, DAC and timer TIM5; all configuration functions were already described in previous chapters.

The endless while loop, lines 26 to 38, is used to calculate the phase angle between signals *XP* and *XR* from results passed by the ADC interrupt routine, and to display the calculated value at the LCD display. The value of a global variable *PhaseDisplay* signals the availability of a new result. When one, the calculation commences.

The phase is represented by a floating point number. This must be converted to integer and fractional component as in previous chapter, and then displayed. The program is given in lines 29 to 35. Some delay is added in line 36 to reduce the number of screen updates.

25. AM radio receiver

The chapter describes the programming of a microcontroller to demodulate a signal from a local radio station. To keep the circuit simple the signal from the local amplitude modulated radio station should be strong, this is the case in the place where the author of this chapter lives. The local radio station transmits at 918 kHz, and a simple LC tuning circuit and antenna suffice to get a signal that can be sampled by the microcontroller. The demodulation is done in real time by the software, and the demodulated signal is sent to an active speaker using a DAC.

25.1. The theory

The block diagram of the AM receiver is depicted in Fig. 25.1. The input signal for the receiver comes from an antenna, but may also come from a suitable amplitude modulated function generator. The input signal gets sampled into *X1*, then comes the block for the removal of a DC component to obtain signal *X2*. Next is a multiplication of the signal *X2* with two sinusoidal signals from a software local oscillator (a DDS generator can be used here); the two output signals from the local oscillator are in quadrature, i.e. their phases differ for 90 degrees. Two low pass filters strip away high frequency components, and the final block calculates the current amplitude of the incoming signal *X1*.

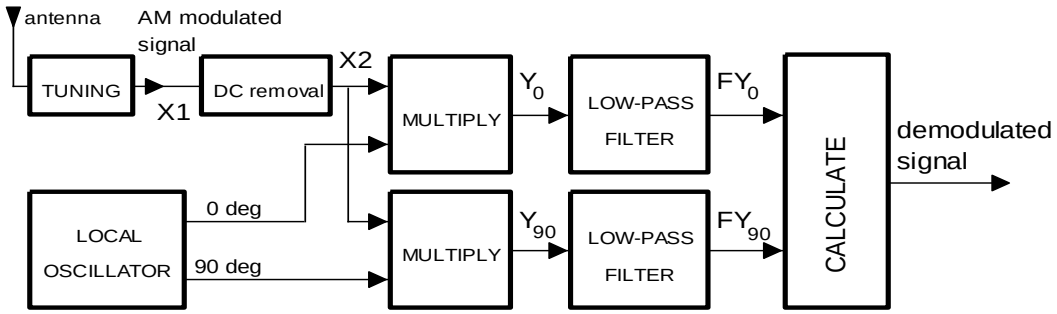


Figure 25.1: The AM receiver: block diagram

Let the input amplitude modulated signal *X1* be:

$$X_1 = A(t)\sin\omega t$$

The frequency ω is the carrier frequency of the received signal, in our case 918 kHz. The amplitude *A(t)* stands for the envelope of the input signal *X1*; since Y1 is amplitude modulated, the envelope depends on time as emphasized by the part in brackets. The envelope represents the signal to be transmitted by the radio, the speech of a narrator, or music. The frequency spectrum of the envelope is limited by the radio transmitter to 4.5 kHz.

The Nyquist criteria states that a signal should be sampled at least twice per period, this would require the sampling frequency of about 2 MHz.

The ADC in the microcontroller can sample at this speed; however, even the microcontroller as fast as the STM32F407 cannot process signals this fast unless we use a trick that is based on an extension of the Nyquist criteria, the trick is called “under-sampling”. The details are explained in [1], here we just present a brief justification of the technique.

Consider a harmonic signal, sampled more than twice per period as the Nyquist criteria requires. An example is given in Figure 25.2; the signal (green) has a frequency of 18 kHz, and the sampling rate is 100 kHz. The green plot is the signal, and the blue dots represent the samples taken by the ADC.

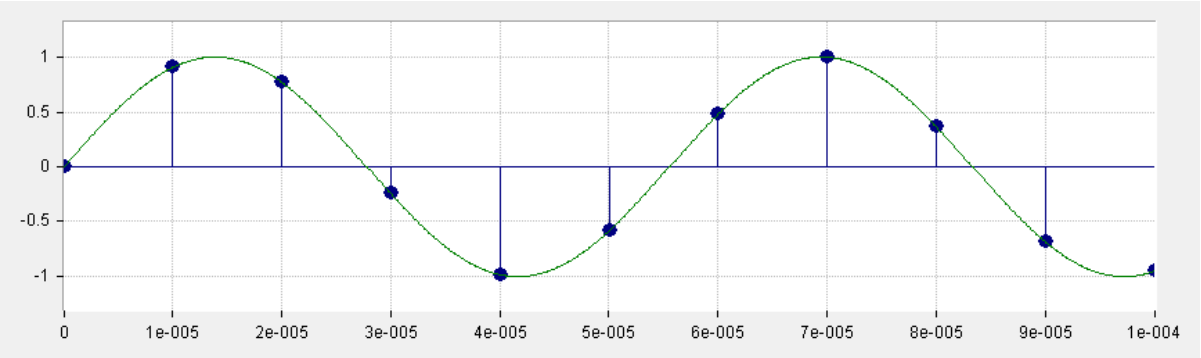


Figure 25.2: Sampling more often than Nyquist criteria requires; horizontal scale in seconds, arbitrary vertical scale

Consider now the same sampling rate and a new input signal with frequency of 318 kHz, which is clearly above the Nyquist requirement. This is shown in Figure 25.3 with the new input signal in red. The sampled values are exactly the same as when sampling the green signal, and the observer is deceived to sample the green signal as in Figure 25.2. There is no way to tell the frequency of the input signal from samples obtained (the blue dots) unless we increase the sample rate.

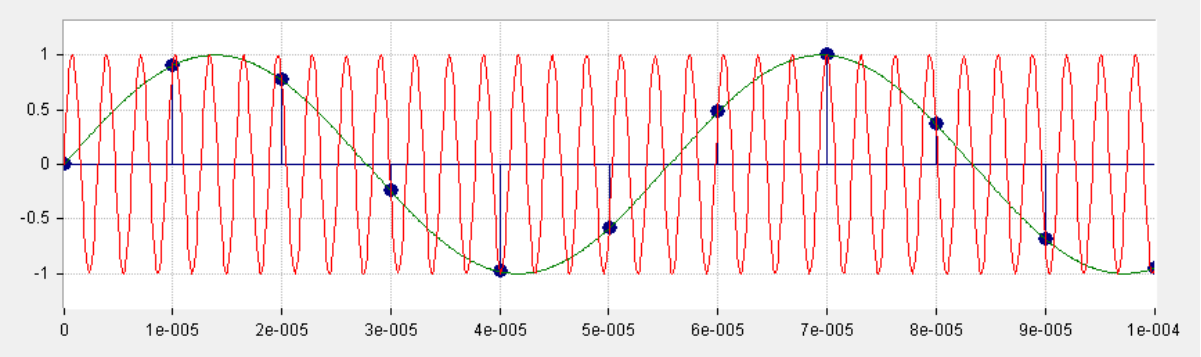


Figure 25.3: Under-sampling: the input signal frequency is 318 kHz

Consider the same sampling rate and an input signal with frequency of 918 kHz, Figure 25.4. Again, the sampled values are the same as when sampling a signal with frequency of 18 kHz!

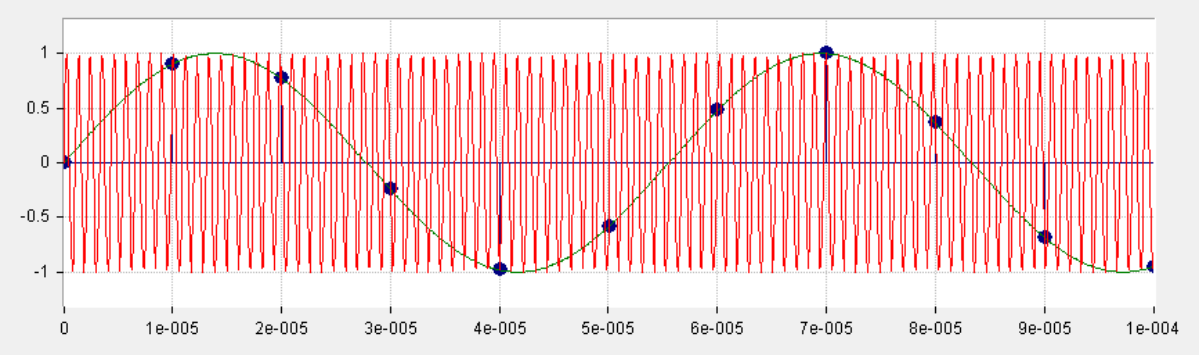


Figure 25.4: Under-sampling: the input signal frequency is 918 kHz

This is why the Nyquist criteria requires to limit the frequency of the input signal to one half of the sampling rate f_s: to be able to determine the frequency of the input signal without any doubts. Also, if there are multiple signals present, and some are above the Nyquist frequency, those appear as having

a frequency in the range from zero to $f_s/2$, and can deceivably change the result of processing. However, if one is not interested in the frequency of the input signal, and one is capable of limiting the range of input frequencies, then one can sample a signal less than twice per period. The signal *X*, which has a frequency f_X anywhere between f_s and $3f_s/2$, will appear as having a frequency of f_x-f_s. Also, when the frequency f_X lies between $2f_s$ and $5f_s/2$, the sampled version will appear to have a frequency of f_x-$2f_s$, Figure 25.5.

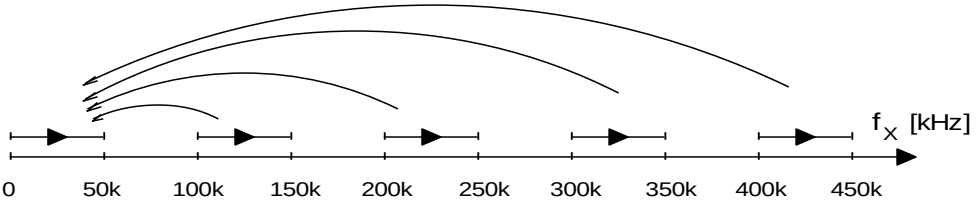


Figure 25.5: Under-sampling regions:
input signal with frequency f_X $(nf_s \le f_X \le (n+0.5)f_s)$ translates as having frequency $f'_X = f_X - nf_s$,
input signal with frequency f_X $((n+0.5)f_s \le f_X \le (n+1)f_s)$ as having a frequency $f'_X = (n+0.5)f_s - f_X$,

We can therefore sample the 918 kHz input signal with a sampling frequency of 100 kHz, and the sampled version of the input signal *X1* will appear to have a frequency of 18 kHz!

There is only one detail still to be considered: the sampling time of the sample & hold (S&H) circuit at the input of the ADC. This circuit is used to make a local copy of the input analog voltage and hold is steady during the conversion. The circuit needs some time to make a local copy, this time is selected by a parameters during the initialization of the ADC. The time to make a copy should be significantly shorter than a period of the input signal, in our case about 1 μs; if the sampling time is long then the circuit is actually averaging the shape of the input signal during the sampling time, and this results in reduced amplitude of the sampled signal. The sampling time of 100 ns is arbitrary the maximum we can afford. In our case the initialization of the ADC uses 'ADC_SampleTime_3Cycles' parameter for function call "ADC_RegularChannelConfig". With the ADC clock set at 30 MHz, one ADC clock cycle takes about 30 ns, and three clock cycles take about 90 ns, which is still sufficient not to reduce the amplitude significantly.

The rest of the data processing is close to the one described in Chapter 23. The sampled input signal *X1* is first stripped of a DC to make *X2*. Since the frequency of the input signal is at 18 kHz, the "DC removal" filter returns almost the same amplitude as the AC signal *X1* has, see the diagram in Figure 24.3.

$$X_2 = A(t)\sin\omega t$$

A local DDS generator provides two signals with frequency$\omega_l = 2\pi \cdot 18\ kHz$, these are phase shifted for 90 degrees. Two multipliers provide signals *Y0* and *Y90*.

$$Y_0 = A(t)\sin\omega t \cdot \sin\omega_l t = \frac{A(t)}{2}\left[\cos(\omega-\omega_l)t - \cos(\omega+\omega_l)t\right]$$

$$Y_{90} = A(t)\sin\omega t \cdot \cos\omega_l t = \frac{A(t)}{2}\left[\sin(\omega-\omega_l)t + \sin(\omega+\omega_l)t\right]$$

Each of these intermediate signals is composed of two parts. Since the frequency of the local oscillator is close to the frequency of the sampled signal, the frequency of the left component $(\omega - \omega_l)$ is almost zero, while the frequency of the right component $(\omega + \omega_l)$ is almost twice the frequency of the local oscillator. Low-pass filters, as the next stage in signal processing, remove both high frequency components.

A brief discussion is in place here. Actually, the left component is multiplied with the envelope signal A(t) which spans up to 4.5 kHz; the product therefore spans up to $4.5\ \mathrm{kHz} + |\omega - \omega_l|$. The low-pass filter should have a corner frequency of at least 4.5 kHz to let the usable components pass. The implemented version of a filter has a corner frequency of 2500 Hz, this affects the frequency spectrum of the received signal, and the music sounds as having a "treble" setting on a regular receiver pushed to a low setting to muffle the high tones. A different set of filter coefficients might be in order here.

The result of the filtering are then *FY0* and *FY90*:

$$FY_0 = \frac{A(t)}{2}\cos(\omega - \omega_l)t \qquad FY_{90} = \frac{A(t)}{2}\sin(\omega - \omega_l)t$$

These two signals contain the information on the amplitude *A(t)* of the signal *X*, and we can calculate the envelope using a simple mathematics:

$$\sqrt{FY_0^2 + FY_{90}^2} = \frac{A(t)}{2} \qquad \rightarrow \qquad A(t) = 2\sqrt{FY_0^2 + FY_{90}^2}$$

Calculating the geometrical sum of signals FY_0 and FY_{90} gives a weighted envelope of the detected signal, this is then sent to an active loudspeaker for listening.

25.2. The implementation of the AM radio receiver

The AM radio receiver project requires an antenna and a tuning circuit, the schematic diagram is given in Figure 25.6. A signal from a function generator can be connected instead, and the generator can be set to generate the AM signal. The output of the tuning circuit is applied to the analog input of the microcontroller, ADC_IN2, where it gets sampled into *X1* and then processed.

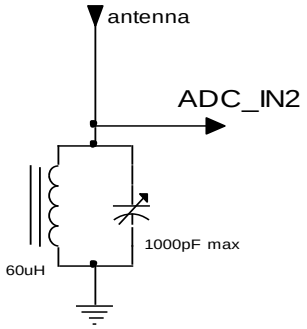


Figure 25.6: The external circuit to the BaseBoard for the AM radio receiver

The program for signal processing is implemented as an interrupt routine, which is driven by timer TIM5, the time interval between successive interrupts is fixed to 10 μs (100 kHz sampling). The program generates signals for local oscillator using the DDS technique, and processes the acquired signals.

The sampling of the input signal is implemented the same way as described in chapters 12 and 13. The timer overflow triggers the ADC to start the conversion, and the end-of-conversion signal from the ADC is used to interrupt the microcontroller and start the interrupt routine ADC_IRQhandler.

The theory behind the AM demodulation requires the calculation of a geometrical sum of two signals. Squaring a variable is fast, but taking a square root of a variable is slow; this takes about 4 μs on the STM32f407 running at full speed. This is just sufficient to perform the complete data processing between two successive samples.

The ADC interrupt function for the AM radio receiver is presented in the listing below.

```
void ADC_IRQHandler(void)  {
  GPIOE->BSRRL = BIT_8;         // about 8 us altogether                                  // 2

  int Spare = ADC1->DR;               // to clear ADC IRQ flag                            // 4
  X1[k] = (ADC2->DR & 0xfff);         // acquire X1                                       // 5
  X2[k] = X1[k] - X1[(k-1) & 63];     // remove DC component                              // 6

  Y0[k]  = X2[k] * Table[  ptrTable >> 4               ];     // multiply with sin        // 7
  Y90[k] = X2[k] * Table[((ptrTable >> 4) + 1024) & 4095];     // multiply with cos        // 8

  float conv = 0;                                              // LP filter (sin)        // 10
```

```
  for (int m = 0; m<5; m++)  conv += a[m] * Y0[(k - m) & 63];                                  // 11
  for (int n = 1; n<5; n++)  conv += b[n] * FY0[(k - n) & 63];                                 // 12
  FY0[k] = conv;                                                                               // 13

  conv = 0;                                                    // LP filter (cos)             // 15
  for (int m = 0; m<5; m++)  conv += a[m] * Y90[(k - m) & 63];                                 // 16
  for (int n = 1; n<5; n++)  conv += b[n] * FY90[(k - n) & 63];                                // 17
  FY90[k] = conv;                                                                              // 18

  float Res = FY0[k] * FY0[k] + FY90[k] * FY90[k];             // add geometrically            // 20
  float ResRes = (float)(sqrt((double)Res));                   // sqrt takes about 4 us !     // 21

  DAC->DHR12R1 = Table[ptrTable >> 4] + 2048;                  // local oscillator out        // 23
  DAC->DHR12R2 = (int)(ResRes / 64 + 128.0);                   // demodulated out             // 24

  ptrTable = (ptrTable + 11796) & 0xffff;                      // next excitation             // 26

  k++; k &= 63;                                                                                // 28
  GPIOE->BSRRH = BIT_8;                                                                        // 29
}
```

The function commences with setting pin 8, port E, high to signal the start of execution, line 2; the same pin is reset when the function ends, line 29. Lines 4 and 5 read the result of conversion from both ADCs and store one of them into circular buffer *X1*, which has 64 elements. The removal of the DC component is performed in line 6, and the result is stored in a circular buffer named *X2*.

The following two lines 7 and 8 implement the multiplication with locally generated signals with frequency of 18 kHz and different phases. The multiplication is followed by filtering for each component separately in lines 10 to 13 and 15 to 18. Both filtered components *FY0* and *FY90* are added geometrically in lines 20 and 21. The resultant envelope is sent to a speaker using a DAC2 in line 24, for testing purposes the signal of local oscillator is available at DAC1.

The frequency of the local oscillator depends on incrementing of the pointer to the table. For frequency of 18 kHz we need to use 11796 to increment the pointer, as implemented in line 26. In practice, any value between 10500 and 13000 will do; values off the correct value only reduce the amplitude of the detected envelope.

The listing of the rest of the program is given below.

```
int main ()  {
  for (ptrTable = 0; ptrTable <= 4095; ptrTable++)                                             // 2
    Table[ptrTable] = (int)(1850.0 * sin((float)ptrTable / 2048.0 * 3.14159265));              // 3

  GPIOEinit ();               ADCinit_T5_CC1_IRQ();          DACinit();                      // 5
  TIM5init_TimeBase_CC1(840);                 // 840 == 10us == 100kHz                       // 6

  while (1) {};                               // endless loop                                // 8
}
```

This listing starts with the inclusion of CMSIS files and the declaration of the variables used; this is not shown in this listing. The function “main()” commences with the initialization of the table for the DDS generator as already explained in chapter 16, and continues to the configuration functions for port E, ADC, DAC and timer TIM5; all configuration functions were already described in previous chapters.

The endless while loop, line 8, is empty; all data processing is performed in the interrupt routine.

[1] Richard G. Lyons: Understanding digital signal processing, second edition, chapter 2