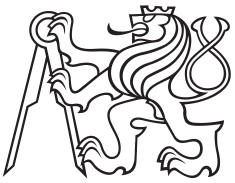
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering

Department of Measurement

Learning and automation GPIO platform

Ondřej Hruška

Supervisor: doc. Ing. Radislav Šmíd, Ph.D.
Field of study: Cybernetics and Robotics
Subfield: Sensors and Instrumentation
2018

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Hruška Ondřej** Personal ID number: **420010**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Measurement**

Study program: **Cybernetics and Robotics**

Branch of study: **Sensors and Instrumentation**

II. Master's thesis details

Master's thesis title in English:

Learning and Automation GPIO Platform

Master's thesis title in Czech:

Výuková a automatizační GPIO platforma

Guidelines:

Design and implement a modular system consisting of a motherboard and additional modules for connecting sensors, actuators and general inputs via I2C, SPI, UART, 1-Wire or other interfaces to the central system via USB, UART, and wireless interfaces. Allow access to built-in processor peripherals such as ADC, DAC, and timers (PWM, frequency measurement). Design a comfortable way to set the configuration without firmware changes. For the designed system, create a service library in C, Python, and MATLAB.

Bibliography / sources:

[1] STMicroelectronics datasheets, http://www.st.com
[2] Ganssle, J.: The Art of Designing Embedded Systems, Elsevier Science, 2008.
[3] Chi, Qingping & Yan, Hairong & Zhang, Chuan & Pang, Zhibo & Da Xu, Li. (2014).: A Reconfigurable Smart Sensor Interface for Industrial WSN in IoT Environment. Industrial Informatics, IEEE Transactions on. 10. 1417-1425. 10.1109/TII.2014.2306798.

Name and workplace of master's thesis supervisor:

doc. Ing. Radislav Šmíd, Ph.D., Department of Measurement, FEL

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **10.01.2018** Deadline for master's thesis submission: _____

Assignment valid until:
by the end of summer semester 2018/2019

doc. Ing. Radislav Šmíd, Ph.D.	Head of department's signature	prof. Ing. Pavel Ripka, CSc.
Supervisor's signature		Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt Student's signature

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 27. května 2018

.....

Acknowledgements

blabla

Abstract

This thesis documents the development of a general purpose software and hardware platform for interfacing low level hardware from high level programming languages and applications run on a PC, using USB and also wirelessly.

The requirements of common engineering tasks and problems occurring in the university environment were evaluated to design an extensible, reconfigurable hardware module that would make a practical, versatile, and low cost tool that in some cases also eliminates the need for professional measurement and testing equipment.

Several hardware prototypes and control libraries in programming languages C and Python have been developed. The Python library additionally integrates with MATLAB scripts. The devices provide access to a range of hardware buses and low level features and can be reconfigured using configuration files stored inside its permanent memory.

Keywords:

Supervisor: doc. Ing. Radislav Šmíd, Ph.D.

Abstrakt

Tato práce popisuje vývoj univerzální softwarové a hardwarové platformy pro přístup k hardwarovým sběrnicím a elektrickým obvodům z prostředí vysokoúrovňových programovacích jazyků a aplikací běžících na PC, a to za využití USB a také bezdrátově.

Byly vyhodnoceny požadavky typických problémů, vyskytujících se v praxi při práci s vestavěnými systémy a ve výuce, pro návrh snadno rozšiřitelného a přenastavitleného hardwarového modulu který bude praktickým, pohodlným a dostupným nástrojem který navíc v některých případech může nahradit profesionální laboratorní přístroje.

Bylo navrženo několik prototypů hardwarových modulů, spolu s obslužnými knihovnami v jazycích C a Python; k modulu lze také přistupovat z prostředí MATLAB. Přístroj umožňuje přístup k většině běžných hardwarových sběrnic a umožňuje také např. měřit frekvenci a vzorkovat či generovat analogové signály.

Klíčová slova:

Překlad názvu: Výuková a automatizační GPIO platforma

Contents

Part III
Implementation

Appendices

Figures

Part I

Introduction

Chapter 1

Motivation

Prototyping, design evaluation, and the measurement of physical properties in experiments make a daily occurrence in the engineering praxis. Those tasks often involve the generation and sampling of electrical signals coming to and from sensors, actuators, and other circuitry.

In the recent years, a wide range of intelligent sensors became available thanks to the drive for miniaturization in the consumer electronics industry. Those devices often provide a sufficient accuracy and precision while keeping the circuit complexity and cost low. In contrast to analog sensors, here the signal conditioning and processing circuits are built into the sensor itself and we access it using a digital connection.

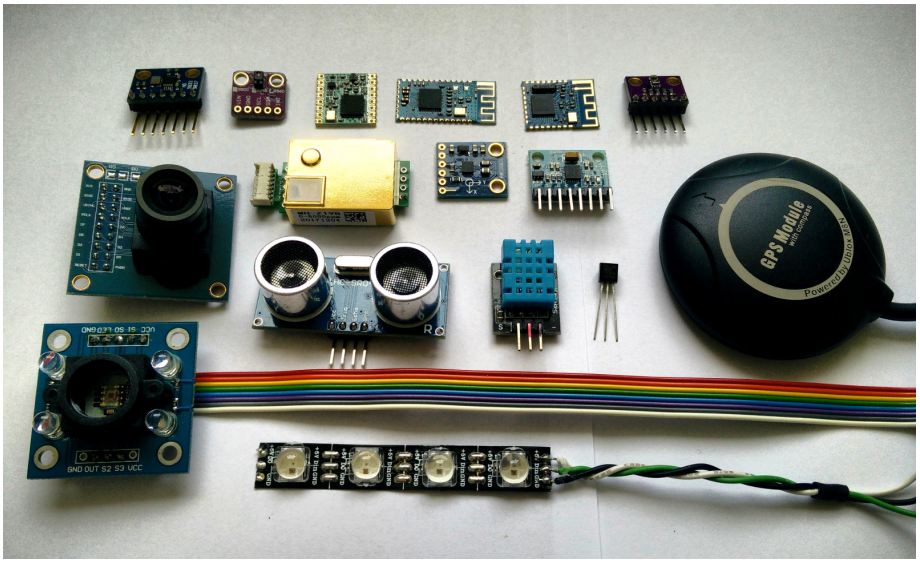


Figure 1.1: A collection of intelligent sensors and devices, most on breadboard adapters: (from top left) a waveform generator, a gesture detector, a LoRa and two Bluetooth modules, an air quality and pressure sensor, a CO_2 sensor, a digital compass, an accelerometer, a GPS module, a camera, an ultrasonic range finder, a humidity sensor, a 1-Wire thermometer, a color detector and an RGB LED strip.

To conduct experiments with those integrated modules, or even just familiarize ourselves with a device before using it in a project, we need a way to easily interact with them. It's also convenient to have a direct access to hardware, be it analog signal sampling, generation, or even just logic level inputs and outputs. However, the drive for miniaturization and

the advent of USB (Universal Serial Bus) lead to the disappearance of low level computer ports, such as the printer port (LPT), that would provide an easy way of doing so.

Today, when one wants to perform measurements using a digital sensor, the usual route is to implement an embedded firmware for a microcontroller that connects to the PC through USB, or perhaps just shows the results on a display. This approach has its advantages, but is time-consuming and requires knowledge entirely unrelated to the measurements we wish to perform. It would be advantageous to have a way to interface hardware without having to burden ourselves with the technicalities of the connection, even at the cost of lower performance compared to a specialized device or a professional tool.

The design and implementation of such a universal instrument is the object of this work. For technical reasons, such as naming the source code repositories, we need a name for the project; it'll be hereafter called *GEX*, a name originating from "GPIO Expander".

1.1 The Project's Expected Outcome

It's been a desire of the author to create an universal instrument connecting low level hardware to a computer for many years, and with this project it is finally being realized. Several related projects approaching this problem from different angles can be found on the internet; those will be presented in chapter 3. This project should not end with yet another tinkering tool that will be produced in a few prototypes and then forgotten. By building an extensible, open-source platform, GEX can become the foundation for future projects which others can expand, re-use and adapt to their specific needs.

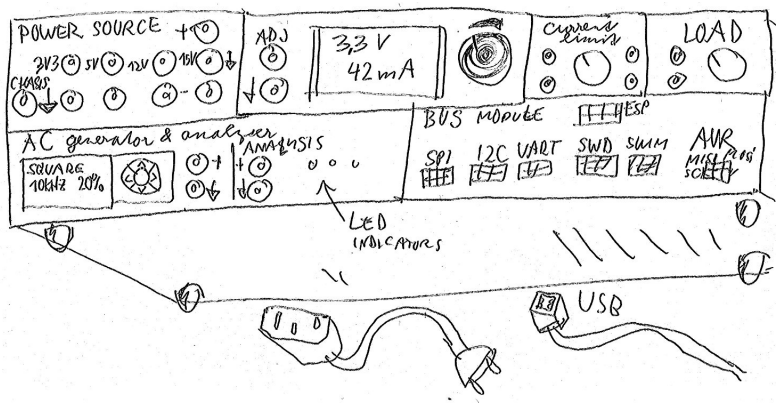


Figure 1.2: An early (2016) sketch of a universal bench device including a power supply, electronic load, a signal generator and a bus module. The bottom half of the panel is in a large part implemented by GEX.

Building on the experience with earlier embedded projects, a STM32 microcontroller shall be used. Those are ARM Cortex M devices with a wide range of hardware peripherals that appear be a good fit for the project. Low-cost evaluation boards are widely available that could be used as a hardware platform instead of developing a custom PCB. In addition, those chips are relatively cheap and popular in the embedded hardware community; there's

a good possibility of the project building a community around it and growing beyond what will be presented in this paper.

Besides the use of existing development boards, custom PCBs will be developed in different form factors. Those could use the Arduino connector or the Raspberry Pi Zero GPIO header (and board shape) to exploit the cases and boxes available for the minicomputer on the market, as well as add-on boards (*shields* and *HATs*).

The possibilities of wireless connection should be evaluated. This feature should make GEX useful e.g. in mobile robotics or when installed in poorly accessible locations.

Chapter 2

Requirement Analysis

We'll now investigate some situations where GEX could be used, to establish its requirements and desired features.

2.1 Desired Features

2.1.1 Interfacing Intelligent Modules

When adding a new digital sensor or a module to a hardware project, we want to test it first, learn how to properly communicate with it and confirm its performance. Based on this evaluation we decide whether the module matches our expectations and learn how to properly connect it, which is needed for a successful PCB layout.

In experimental setups, this may be the only thing we need. Data can readily be collected after just connecting the module to a PC, same as commanding motor controllers or other intelligent devices.

A couple well known hardware buses have established themselves as the standard ways to interface digital sensors and modules: SPI, I2C and UART are the most used ones, often accompanied by a few extra GPIO lines such as Reset, Chip Enable, Interrupt. There are exceptions where silicon vendors have developed proprietary communication protocols that are still used, either for historical reasons or because of their specific advantages. An example is the 1-Wire protocol used by digital thermometers.

Moving to industrial and automotive environments, we can encounter various fieldbuses, Ethernet, CAN, current loop, HART, LIN, DALI, RS485 (e.g. Modbus), mbus, PLCBUS and others. Those typically use transceiver ICs and other circuitry, such as TVS, discrete filters, galvanic isolation etc. They could be supported using add-on boards and additional firmware modules handling the protocol. For simplicity and to meet time constraints, the development of those boards and modules will be left for future expansions of the project.

2.1.2 Analog Signal Acquisition

Sometimes it's necessary to use a traditional analog sensor, capture a transient waveform or to just measure a voltage. GEX was meant to focus on digital interfaces, however giving it this capability makes it much more versatile. Nearly all microcontrollers include an

analog-digital converter which we can use to measure input voltages and, paired with a timer, to records signals varying in time.

Certain tasks, such as capturing transient effects on a thermocouple when inserted into a flame (an example from developing fire-proof materials) demand level triggering similar to that of oscilloscopes. The converter continuously measures the input voltage and a timed capture starts only after a set threshold is exceeded. This can be accompanied by a pre-trigger feature where the timed capture is continuously running and the last sample is always compared with the threshold, recording a portion of the historic records together with the following samples.

2.1.3 Analog Signal Output

An analog signal can not only be measured, but it's often necessary to also generate it. This could serve as an excitation signal for an experiment, for instance to measure the characteristic curves of a diode or a transistor. Conveniently, we can at the same time use GEX's analog input to record the output.

Generating an analog signal is possible using a pulse-width modulation (PWM) or by a dedicated digital-analog converter included in many microcontrollers. Higher frequencies or resolution can be achieved with a dedicated external IC.

2.1.4 Logic Level Input and Output

We've covered some more advanced features, but skipped the simplest feature: a direct access to GPIO pins. Considering the latencies of USB and the PC's operating system, this can't be reliably used for "bit banging", however we can still accomplish a lot with just changing logic levels - e.g. to control character LCDs, or emulate some interfaces that include a clock line, like SPI. As mentioned in 2.1.1, many digital sensors and modules use plain GPIOs in addition to the communication bus for out-of-band signaling or features like chip selection or reset.

2.1.5 Pulse Generation and Measurement

Some sensors have a variable frequency or a pulse-width modulated (PWM) output. To capture those signals and convert them to a more useful digital value, we can use the external input functions of a timer/counter in the microcontroller. Those timers have many possible configurations and can also be used for pulse counting or a pulse train generation.

2.2 Host Computer Connection

2.2.1 Communication Interface

USB shall be the primary way of connecting the module to a host PC. Thanks to USB's flexibility, it can present itself as any kind of device or even multiple devices at once.

The most straightforward method of interfacing the board is by passing binary messages in a fashion similar to USART (and plain UART can be available as well). We'll need a duplex connection to enable command confirmations, query-type commands and asynchronous event reporting. This is possible either using a "Virtual COM port" driver (the CDC/ACM USB class), or through a raw access to the corresponding USB endpoints. Using a raw access avoids potential problems with the operating system's driver interfering or not recognizing the device correctly; on the other hand, having GEX appear as a serial port makes it easier to integrate into existing platforms that have a good serial port support (such as National Instruments LabWindows CVI or MATLAB).

A wireless attachment is also planned; after establishing a connection, the two-way radio link should work transparently, in a similar manner to the UART or USB connection.

2.2.2 Configuration Files

The module must be easily reconfigurable. Given the settings are almost always going to be tied on the connected external hardware, it would be practical to have an option to store them permanently in the microcontroller's non-volatile memory.

We can load those settings into GEX using the serial interface, which also makes it possible to reconfigure it remotely when the wireless connection is used. With USB, we can additionally make the board appear as a mass storage device and expose the configuration as text files. This approach, inspired by ARM mbed's mechanism for flashing firmware images to development kits, avoids the need to create a configuration GUI, instead using the PC OS's built-in applications like File Explorer and Notepad. We can expose additional information, such as a README file with instructions or a pin-out reference, as separate files on the virtual disk.

2.3 An Overview of Planned Features

Let's now summarize the features we wish to support in the GEX firmware, based on the preceding discussion:

- **Hardware interfacing functions**
  - I/O pin direct access (read, write), pin change interrupt
  - Analog input: voltage measurement, sampled capture
  - Analog output: static level, waveform generation
  - Frequency, duty cycle, pulse length measurement
  - Single pulse and PWM generation
  - SPI, I^2C, UART/USART
  - Dallas 1-Wire
  - NeoPixel (addressable LED strips)

- **Communication with the host computer**
  - USB connection as virtual serial port or direct endpoint access
  - Connection using plain UART
  - Wireless attachment
- **Configuration**
  - Fully reconfigurable, temporarily or permanently
  - Settings stored in INI files
  - File access through the communication API or using a virtual mass storage

2.4 Microcontroller Selection

As discussed in section 1.1, this project will be based on microcontrollers from the STM32 family. The STM32F072 model was selected for the initial hardware and firmware design due to it's low cost, advanced peripherals and the availability of development boards. The firmware can be ported to other MCUs later (e.g. to STM32L072, STM32F103 or STM32F303).

The STM32F072 is a Cortex M0 device with 128 KiB of flash memory, 16 KiB of RAM and running at 48 MHz. It is equipped with a USB Full Speed peripheral block, a 12-bit ADC and DAC, a number of general purpose timers/counters, SPI, I^2C, and USART peripherals, among others. It supports crystal-less USB, using the USB SOF packet for synchronization of the internal 48 MHz RC oscillator; naturally, a real crystal resonator will provide better timing accuracy.

To effectively utilize the time available for this work, only the STM32F072 firmware will be developed while making sure the planned expansion is as straightforward as possible.

2.5 Form Factor Considerations

While the GEX firmware can be used with existing evaluation boards from ST Microelectronics (see figure 2.1 for an example of one such board), we wish to design and realize a few custom hardware prototypes that will be smaller and more convenient to use.

Three possible form factors are drawn in figure 2.2. The use of a common connector layout and pin assignments, here Arduino and Raspberry Pi, makes it possible to reuse add-on boards from those platforms. When we copy the physical form factor of another product, in this example the Raspberry Pi Zero, we can further take advantage of existing enclosures designed for it.

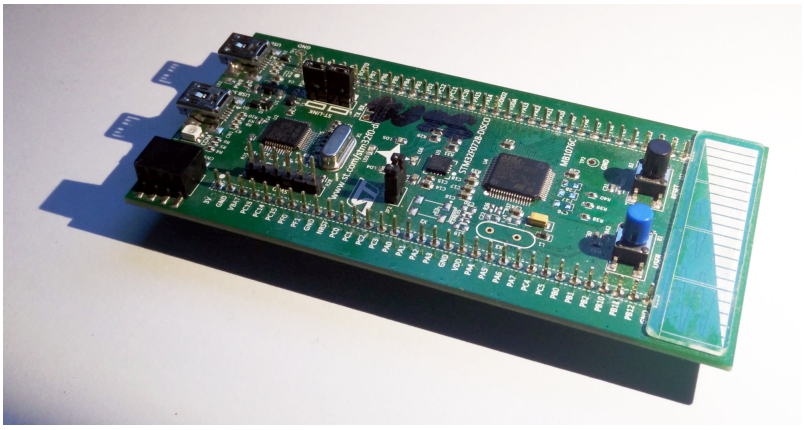


Figure 2.1: A Discovery development board with the STM32F072 microcontroller

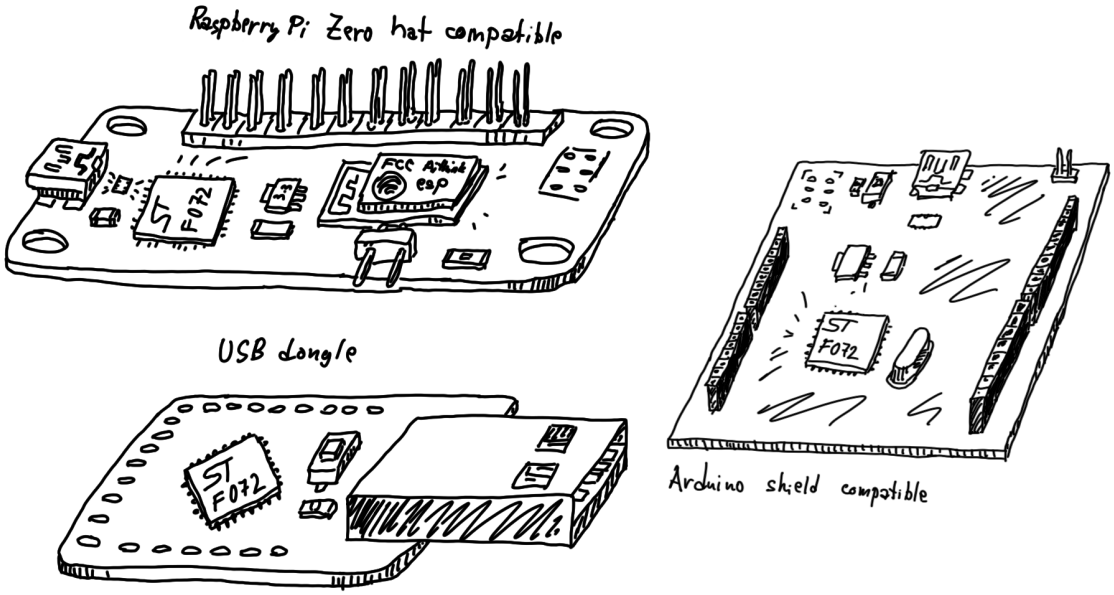


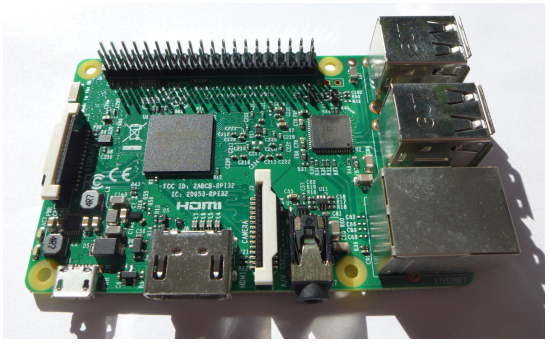
Figure 2.2: A sketch of three possible form factors for the GEX hardware prototype. Note the ESP8266 module which was considered as an option for wireless access but was eventually not used due to it's high current usage, unsuitable for battery operation.

Chapter 3

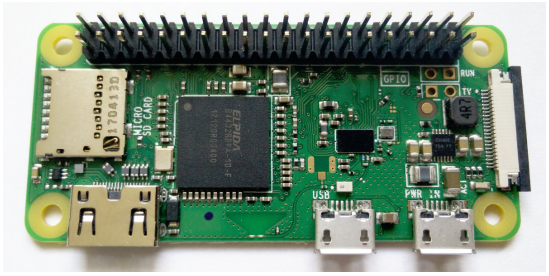
Existing Solutions

The idea of making it easier to interact with low level hardware from a PC is not new. Several solutions to this problem have been developed, each with its own advantages and drawbacks. Some examples will be presented in this chapter.

3.1 Raspberry Pi



(a) : Raspberry Pi 3 Model B



(b) : Raspberry Pi Zero W

Figure 3.1: Raspberry Pi minicomputers

The Raspberry Pi's GPIO header, a row of pins which can be directly controlled by user applications running on the minicomputer, was one of the inspirations behind GEX. It can be controlled using C and Python (among others) and offers general purpose I/O, SPI, I2C, UART and PWM, with other protocols easy to emulate thanks to the high speed of the system processor.

The Raspberry Pi is used in schools as a low-cost PC alternative that encourage students' interest in STEM (Science, Technology, Engineering and Mathematics). The board is often built into more permanent projects that make use of its powerful processor, such as wildlife camera traps, fish feeders etc.

The Raspberry Pi could be used for the same quick evaluations or experiments we want to perform with GEX, however they would either have to be performed directly on the mini-computer, with an attached monitor and a keyboard, or use some form of remote access (e.g. SSH, screen sharing).

3.2 Bus Pirate

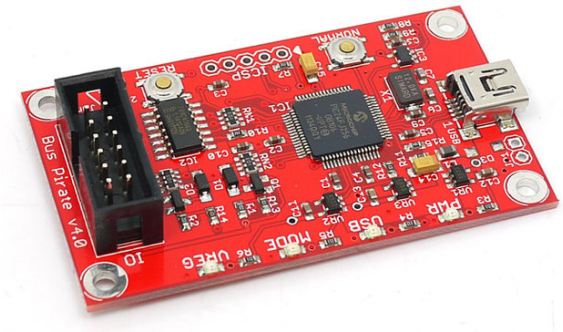


Figure 3.2: Bus Pirate v.4 (picture by *Seeed Studio*)

Bus Pirate, a project by Ian Lesnet, is a USB-attached device providing access to hardware interfaces like SPI, I^2C, USART and 1-Wire, as well as frequency measurement and direct pin access. The board aims to make it easy for users to familiarize themselves with new chips and modules; it also provides a range of programming interfaces for flashing microcontroller firmwares and memories. It communicates with the PC using a FTDI USB-serial bridge.

Bus Pirate is open source and is, in its scope, similar to GEX. It can be scripted and controlled from the PC, connects to USB and provides a wide selection of hardware interfaces.

The board is based on a PIC16 microcontroller running at 32 MHz. Its analog/digital converter (ADC) only has a resolution of 10 bits (1024 levels). There is no digital/analog converter (DAC) available on the chip, making applications that require a varied output voltage more difficult. Another limitation of the board is its low number of GPIO pins, which may be insufficient for certain applications. The Bus Pirate is available for purchase at around 30 USD, a price comparable to some Raspberry Pi models.

3.3 Professional DAQ Modules

Various professional tools that would fulfill our needs exist on the market, but their high price makes them inaccessible for users with a limited budget, such as hobbyists or students who would like to keep such a device for personal use. An example is the National Instruments (NI) "I²C/SPI Interface Device" which also includes several GPIO lines, the NI USB DAQ module, or some of the Total Phase I²C/SPI gadgets (figure 3.3). The performance GEX can provide may not always match that of those professional tools, but in many cases it'll be a sufficient substitute at a fraction of the cost.

(a) : NI I²C/SPI Interface Device

(b) : NI USB DAQ module

(c) : Total Phase SPI/I²C Host "Aardwark"

Figure 3.3: An example of professional tools that GEX could replace in less demanding scenarios *(pictures from National Instruments and Total Phase marketing materials)*

Part II

Theoretical Background

Chapter 4

Universal Serial Bus

This chapter presents an overview of the *Universal Serial Bus* (USB) *Full Speed* interface, with focus on the features used in the GEX firmware. USB is a versatile but complex interface, thus explaining it in its entirety is beyond the scope of this text. References to external material which explains the protocol in greater detail will be provided where appropriate.

add those refs

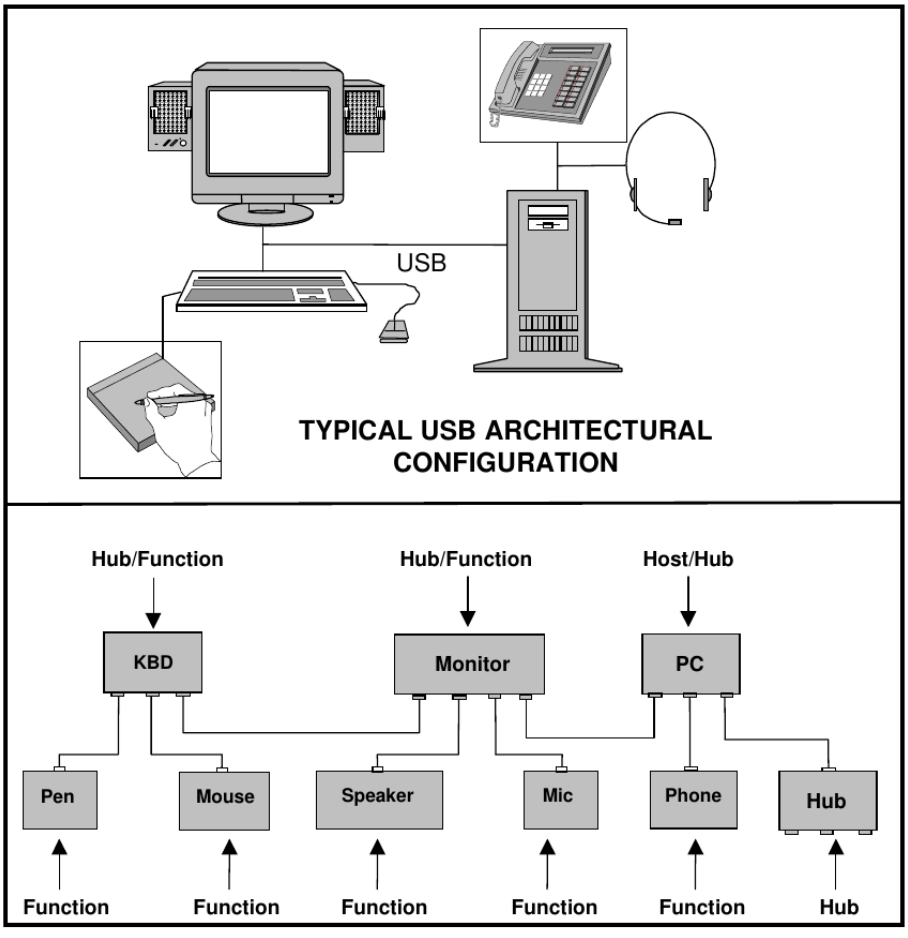


Figure 4.1: A diagram from the USB specification rev. 1.1 showing the hierarchical structure of the USB bus; The PC (Host) controls the bus and initiates all transactions.

4.1 Basic Principles and Terminology

review and correct inaccuracies

USB is a hierarchical bus with a single master (*host*) and multiple slave devices. A USB device that provides functionality to the host is called a *function*. Communication between the host and a function is organized into virtual channels called *pipes*. Each pipe is identified by an *endpoint* number.

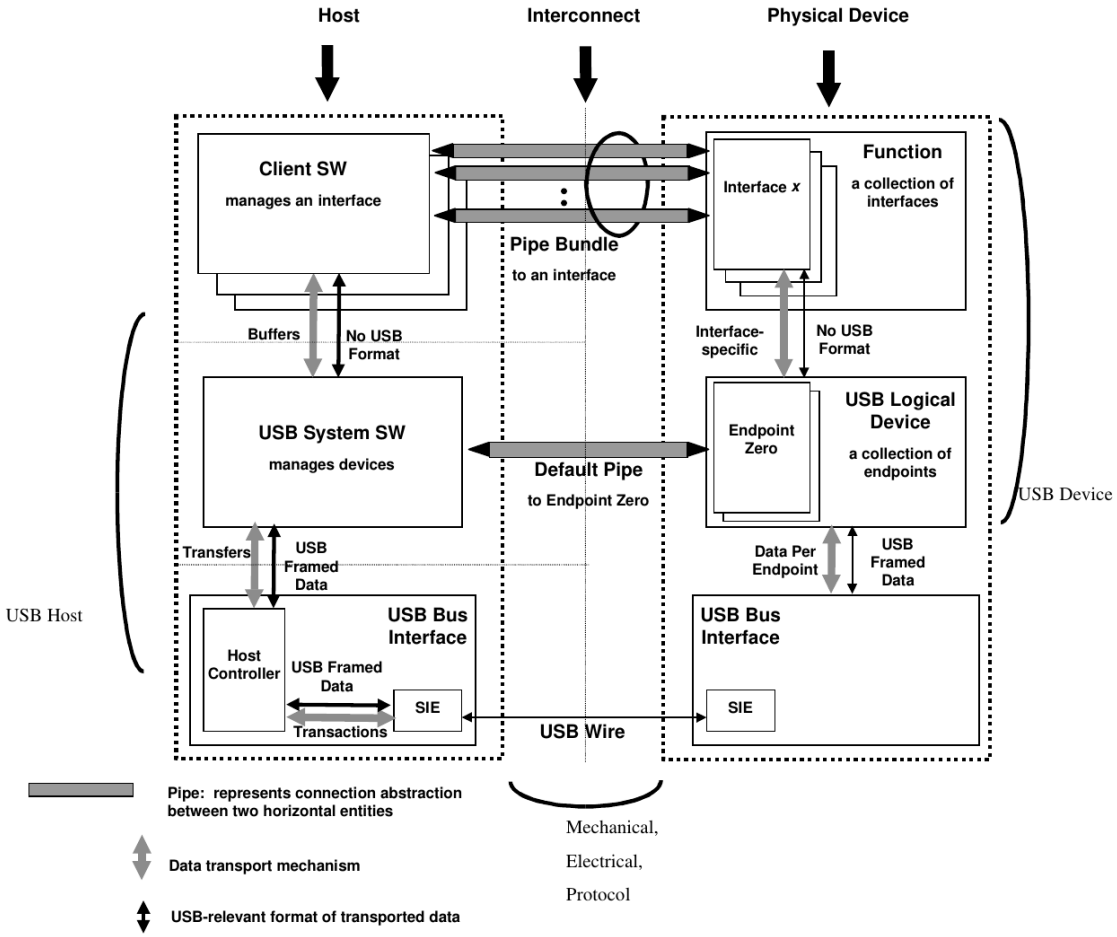


Figure 4.2: A detailed view of the host-device connection (*USB specification rev. 1.1*)

Endpoints can be either unidirectional or bidirectional; the direction from the host to a function is called *OUT*, the other direction (function the host) is called IN. A bidirectional endpoint is technically composed of a IN and OUT endpoint with the same number. All transactions (both IN and OUT) are initiated by the host; functions have to wait for their turn. Endpoint 0 is bidirectional, always enabled, and serves as a *control endpoint*. The host uses the control endpoint to read information about the device and configure it as needed.

There are four types of transfers: control, bulk, isochronous, and interrupt. Each endpoint is configured for a fixed transfer type.

- *Control* - initial configuration after device plug-in; also used for other aplication-specific control messages that can affect other pipes.
- *Bulk* - used for burst transfers of large messages, commonly e.g. for mass storage devices
- *Isochronous* - streaming with guaranteed low latency; designed for audio or video streams where some data loss is preferred over stuttering
- *Interrupt* - low latency short messages, used for human interface devices like mice and keyboards

The endpoint transfer type and other characteristics, together with other information about the device, such as the serial number, are defined in a *descriptor table*. This is a tree-like binary structure defined in the function's memory. The descriptor table is loaded by the host to learn about the used endpoints and to attach the right driver to it.

The function's endpoints are grouped into *interfaces*. An interface describes a logical connection of endpoints, such as the reception and transmission endpoint that belong together. An interface is assigned a *class* defining how it should be used. Standard classes are defined by the USB specification to provide a uniform way of interfacing devices of the same type, such as human-interface devices (mice, keyboards, gamepads) or mass storage devices. The use of standard classes makes it possible to re-use the same driver software for devices from different manufacturers. The class used for the GEX's "virtual COM port" function was originally meant for telephone modems, a common way of connecting to the Internet at the time the first versions of USB were developed. A device using this class will show as `/dev/ttyACM0` on Linux and as a COM port on Windows, provided the system supports it natively or the right driver is installed.

add reference to the document

```
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass          239 Miscellaneous Device
  bDeviceSubClass         2
  bDeviceProtocol         1 Interface Association
  bMaxPacketSize0        64
  idVendor           0x0483 STMicroelectronics
  idProduct          0x572a
  bcdDevice            0.01
  iManufacturer           1 MightyPork
  iProduct                2 GEX
  iSerial                 3 0029002F-42365711-32353530
  bNumConfigurations      1
  Configuration Descriptor:
    bLength                 9
    bDescriptorType         2
    wTotalLength           98
    bNumInterfaces          3
    bConfigurationValue     1
    iConfiguration          0
    bmAttributes         0x80
      (Bus Powered)
    MaxPower              500mA
    Interface Descriptor:
      bLength                 9
      bDescriptorType         4
      bInterfaceNumber        0
      bAlternateSetting       0
      bNumEndpoints           2
      bInterfaceClass         8 Mass Storage
      bInterfaceSubClass      6 SCSI
      bInterfaceProtocol     80 Bulk-Only
      iInterface              4 Settings VFS
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x81  EP 1 IN
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0040  1x 64 bytes
        bInterval               0
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x01  EP 1 OUT
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0040  1x 64 bytes
        bInterval               0
    Interface Association:
      bLength                 8
      bDescriptorType        11
      bFirstInterface         1
      bInterfaceCount         2
      bFunctionClass          2 Communications
      bFunctionSubClass       2 Abstract (modem)
      bFunctionProtocol       1 AT-commands (v.25ter)
      iFunction               5 Virtual Comport ACM
Interface Descriptor:
  bLength                 9
  bDescriptorType         4
  bInterfaceNumber        1
  bAlternateSetting       0
  bNumEndpoints           1
  bInterfaceClass         2 Communications
  bInterfaceSubClass      2 Abstract (modem)
  bInterfaceProtocol      1 AT-commands (v.25ter)
  iInterface              5 Virtual Comport ACM
  CDC Header:
    bcdCDC               1.10
  CDC Call Management:
    bmCapabilities       0x00
    bDataInterface          2
  CDC ACM:
    bmCapabilities       0x06
      sends break
      line coding and serial state
  CDC Union:
    bMasterInterface        1
    bSlaveInterface         2
  Endpoint Descriptor:
    bLength                 7
    bDescriptorType         5
    bEndpointAddress     0x83  EP 3 IN
    bmAttributes            3
      Transfer Type            Interrupt
      Synch Type               None
      Usage Type               Data
    wMaxPacketSize     0x0008  1x 8 bytes
    bInterval             255
Interface Descriptor:
  bLength                 9
  bDescriptorType         4
  bInterfaceNumber        2
  bAlternateSetting       0
  bNumEndpoints           2
  bInterfaceClass        10 CDC Data
  bInterfaceSubClass      0
  bInterfaceProtocol      0
  iInterface              6 Virtual Comport CDC
  Endpoint Descriptor:
    bLength                 7
    bDescriptorType         5
    bEndpointAddress     0x02  EP 2 OUT
    bmAttributes            2
      Transfer Type            Bulk
      Synch Type               None
      Usage Type               Data
    wMaxPacketSize     0x0040  1x 64 bytes
    bInterval               0
  Endpoint Descriptor:
    bLength                 7
    bDescriptorType         5
    bEndpointAddress     0x82  EP 2 IN
    bmAttributes            2
      Transfer Type            Bulk
      Synch Type               None
      Usage Type               Data
    wMaxPacketSize     0x0040  1x 64 bytes
    bInterval               0
```

Figure 4.3: USB descriptors of a GEX prototype obtained using `lsusb -vd vid:pid`

4.2 USB Physical Layer

USB uses differential signaling with NRZI encoding (*Non Return to Zero Inverted*, fig. 4.4) and bit stuffing. The encoding, together with frame formatting, checksum verification, retransmission, and other low level aspects of the USB connection are entirely handled by the USB block in the microcontroller's silicon; normally we do not need to worry about those details. What needs more attention are the electrical characteristics of the bus, which need to be understood correctly for a successful schematic and PCB design.

The USB cable contains 4 conductors:

- V_{BUS} (+5 V)
- D+
- D–
- Ground

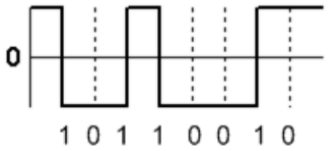


Figure 4.4: NRZI encoding example

The data lines, D+ and D–, are also commonly labeled DP and DM. This differential pair should be routed in parallel and kept at approximately the same length.

USB revisions are, where possible, backwards compatible, often even keeping the same connector shape. The bus speed is negotiated by the device using a 1.5 kΩ pull-up resistor to 3.3 V on one of the data lines: for Full Speed, D+ is pulled high (fig. 4.5), for Low Speed it's on D–. The polarity of the differential signals is inverted depending on the used speed. Some microcontrollers integrate the correct pull-up resistor inside the USB block (including out STM32F072), removing the need for an external resistor.

When a function wants to be re-enumerated by the host, which is needed to reload the descriptors and re-attach the correct drivers, it can momentarily remove the pull-up resistor, which the host will interpret as if the device was plugged out. With an internal pull-up this can be done by flipping a bit in a control register. An external resistor can be connected through a transistor controlled by a GPIO pin.

https://www.eevblog.com/forum/projects/driving-the-1k5-usb-pull-up-resistor-on-d/

http://www.beyondlogic.org/usbnutshell/usb2.shtml

The V_{BUS} line supplies power to *bus-powered* devices. *Self-powered* devices can leave this pin unconnected and instead use an external power supply. The maximal current drawn from the V_{BUS} line is configured using a descriptor and should not be exceeded, but experiments suggest this is often not enforced.

4.3 USB Classes

This section explains the Mass Storage class and the CDC/ACM class that are used in the GEX firmware. A list of all standard classes with a more detailed explanation can be found on the USB.org website at `http://www.usb.org/developers/defined_class`.

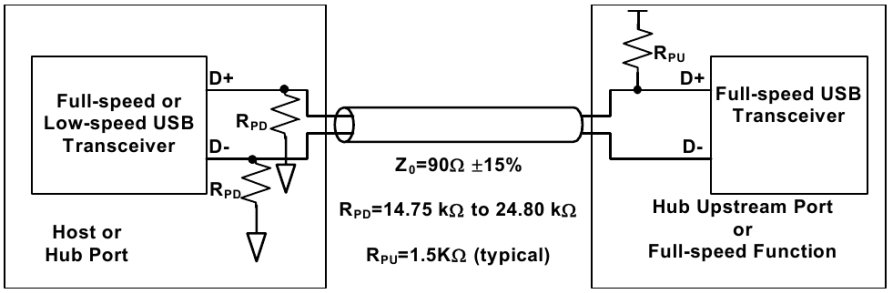


Figure 4.5: Pull-up and pull-down resistors of a Full Speed function, as prescribed by the USB specification rev. 2.0

4.3.1 Mass Storage Class

The Mass Storage class (MSC) is supported by all modern operating systems (MS Windows, MacOS, GNU/Linux, FreeBSD etc.) to support thumb drives, external disks, memory card readers and other storage devices.

references

The MSC specification defines multiple *transport protocols* that can be selected using the descriptors. For it's simplicity, the *Bulk Only Transport* (BOT) will be used. BOT uses two bulk endpoints for reading and writing blocks of data and for the exchange of control commands and status messages. For the device to be recognized by the operating system, it must also implement a *command set*. Most mass storage devices use the *SCSI Transparent command set* [1]. The command set's commands let the host read information about the attached storage, such as its capacity, and check for media presence and readiness to write or detach. This is used e.g. for the "Safely Remove" function which checks that all internal buffers have been written to Flash.

links

The MSC class together with the SCSI command set are implemented in a USB Device library provided by ST Microelectronics. The library also includes a basic CDC/ACM implementation (see below).

In order to emulate a mass storage device without having a physical storage medium, we need to generate and parse the filesystem on-the-fly as the host OS tries to access it. This will be discussed in chapter 6.

4.3.2 CDC/ACM Class

Historically meant for modem communication, this class is now the de facto standard way of making USB devices appear as serial ports on the host OS. The CDC (*Communication Device Class*) uses three endpoints: bulk IN and OUT, and an interrupt endpoint.

[1] To confirm this assertion, the descriptors of five thumb drives and an external hard disk were analyzed using `lsusb`. All but one device used the SCSI command set, one (the oldest thumb drive) used *SFF-8070i*. A list of possible command sets can be found in TODO (usb spec overview)

The interrupt endpoint is used for control commands and notifications while the bulk endpoints are used for useful data. ACM stands for *Abstract Control Model* and it's a CDC's subclass that defines the control messages format. Since we don't use a physical UART and the line is virtual both on the PC and in the end device, the control commands can be ignored.

verify this vvv

An interesting property of this class is that the bulk endpoints transport raw data without any wrapping frames. By changing the device class in the descriptor table to 255 (*Vendor Specific Class*), we can retain the messaging functionality of the designated endpoints and access the device directly using e.g. libUSB, while the OS will ignore it and won't try to attach any driver that could interfere otherwise. The same trick can be used to hide the mass storage class when not needed.

4.3.3 Interface Association: Composite Class

Since it's creation, the USB specification expected that each function will have only one interface enabled at a time. After it became apparent that there is a need for having multiple unrelated interfaces work in parallel, a workaround called the *Interface Association Descriptor* (IAD) was introduced. IAD is an entry in the descriptor table that defines which interfaces belong together and should be handled by the same software driver.

To use the IAD, the function's class must be set to 239 (EFh), subclass 2 and protocol 1, so the OS knows to look for the presence of IADs before binding drivers to any interfaces.

In GEX, the IAD is used to tie together the CDC and ACM interfaces while leaving out the MSC interface which should be handled by a different driver. To make this work, a new *composite class* had to be created as a wrapper for the library-provided MSC and CDC/ACM implementations.

Chapter 5

FreeRTOS

FreeRTOS is a free, open-source real time operating system kernel that has been ported to over 30 microcontroller architectures. The kernel provides a scheduler and implements queues, semaphores and mutexes that are used for message passing between concurrent tasks and for synchronization. FreeRTOS is compact designed to be easy to understand; it's written in C with the exception of some architecture-specific routines that use assembly.

FreeRTOS is used in GEX for its synchronization objects and queues that make it easy to safely pass messages from USB interrupts to a working thread that processes them and sends back responses. Similar mechanism is used to handle external interrupts.

5.1 Basic FreeRTOS Concepts and Functions

5.1.1 Tasks

Threads in FreeRTOS are called *tasks*. Each task is assigned a memory area to use as its stack space, and a structure with it's name, saved context and other metadata used by the kernel. A context includes the program counter, stack pointer and other register values. Task switching is done by saving and restoring this context by manipulating the values on stack before leaving and interrupt.

At start-up the firmware initializes the kernel, registers tasks to run and starts the scheduler. From this point onward the scheduler is in control and runs the tasks using a round robin scheme. Which task should run is primarily determined by their priority numbers, but there are other factors. FreeRTOS supports both static and dynamic object creation, including registering new tasks at run-time.

Task Run States

Tasks can be in one of four states: Suspended, Ready, Blocked, Running. The Suspended state does not normally occur in a task's life cycle, it's entered and left using API calls on demand. A task is in the Ready state when it can run, but is currently paused because a higher priority task is running. It enters the Running state when the scheduler switches to it. A Running task can wait for a synchronization object (e.g. a mutex) to be available. At this point it enters a Blocked state and the scheduler runs the next Ready task. When no tasks can run, the Idle Task takes control; it can either enter a sleep state to save power, or wait in an infinite loop until another task is available.

Task Switching and Interrupts

Task switching occurs periodically in a SysTick interrupt, usually every 1 ms. After one tick of run time, the running task is paused (enters Ready state), or continues to run if no higher priority task is available. If a high priority task waits for an object and this is made available in an interrupt, the previously running task is paused and the waiting task is resumed immediately (enters the Running state).

Only a subset of the FreeRTOS API can be accessed from interrupt routines, for example it's not possible to use the delay function or wait for an object with a timeout, because the SysTick interrupt which increments the tick counter has the lowest priority and couldn't run. This is by design to prevent unexpected context switching in nested interrupts.

FreeRTOS uses a *priority inheritance* mechanism to prevent situations where a high priority task waits for an object held by a lower priority task (called *priority inversion*). The blocking task's priority is temporarily raised to the level of the blocked high priority task so it can finish faster and release the held object. Its priority is then degraded back to the original value. When the lower priority task itself is blocked, the same process can be repeated.

5.1.2 Synchronization Objects

FreeRTOS provides binary and counting semaphores, mutexes and queues.

Binary semaphores can be used for task notifications, e.g. a task waits for a semaphore to be set by an interrupt when a byte is received on the serial port. This makes the task Ready and if it has a higher priority than the previously running task, it's immediately resumed to process the event.

Counting semaphores are used to represent available resources. A pool of resources (e.g. DMA channels) is accompanied by a counting semaphore, so that tasks can wait for a resource to become available in the pool and then subtract the semaphore value. After finishing with a resource, the semaphore is incremented again and another task can use it.

Mutexes, unlike semaphores, must be taken and released in the same thread (task). They're used to guard exclusive access to a resource, such as transmitting on the serial port. When a mutex is taken, a task that wishes to use it enters Blocked state and is resumed once the mutex becomes available and it can take it.

Queues are used for passing messages between tasks, or from interrupts to tasks. Both sending and receiving queue messages can block until the operation becomes possible.

In GEX, mutexes and semaphores are used for sending messages to the PC, and a queue is used for processing received bytes and to send messages from interrupts, because it's not possible to block on a mutex or semaphore while inside an interrupt routine.

Chapter 6

The FAT16 File System and Its Emulation

A file system (FS) is used by GEX to provide a comfortable access to the configuration files. By emulating a Mass Storage USB device, the module appears as a thumb drive on the host PC, and the user can edit its configuration using their preferred text editor. The FAT16 file system was selected for its simplicity and a good cross-platform support.

Three variants of the FAT (File Allocation Table) file system exist: FAT12, FAT16, and FAT32. FAT12 was used on floppy disks and it is similar to FAT16, except for additional size constraints and a FAT entry packing scheme. FAT16 and FAT32 are FAT12's later developments from the time when hard disks became more common and the old addressing scheme couldn't support their larger capacity.

This chapter will explain the structure of FAT16 and the challenges faced when trying to emulate it without a physical data storage.

6.1 The General Structure of the FAT File System

The storage medium is organized into *sectors* (or *blocks*), usually 512 bytes long. Those are the smallest addressing unit in the disk structure. The disk starts with a *boot sector*, also called *master boot record* (MBR). That is followed by optional reserved sectors, one or two copies of the file allocation table, and the root directory. All disk areas are aligned to a sector boundary:

Disk area	**Size / Notes**
Boot sector	1 sector
Reserved sectors	optional
FAT 1	1 or more sectors, depends on disk size
FAT 2	optional, a back-up copy of FAT 1
Root directory	1 or more sectors
Data area	Organized in *clusters*

Table 6.1: Areas of a FAT-formatted disk

6.1.1 Boot Sector

This is a 1-sector structure which holds the OS bootstrap code for bootable disks. The first 3 bytes are a jump instruction to the actual bootstrap code located later in the sector. What matters to us when implementing the file system is that the boot sector also contains data fields describing how the disk is organized, what file system is used, who formatted it, etc. The size of the FAT and the root directory is defined here. The exact structure of the boot sector can be found in XXXor in the attached GEX source code.

add ref link

6.1.2 File Allocation Table

The data area of the disk is organized in clusters, logical allocation units composed of groups of sectors. The use of a larger allocation unit allows the system to use shorter addresses and thus support a larger disk capacity.

The FAT acts as a look-up table combined with linked lists. In FAT16, it is organized in 16-bit fields, each corresponding to one cluster. The first two entries in the allocation table are reserved and hold special values set by the disk formatter and the host OS: a "media descriptor" 0xFFF8 and a "clean/dirty flag" 0xFFFF/0x3FFF.

Files can span multiple clusters; each FAT entry either holds the address of the following file cluster, or a value with a special meaning:

- 0x0000 - free cluster
- 0xFFFF - last cluster of the file (still including file data!)
- 0xFFF7 - bad cluster

The bad cluster mark, 0xFFF7, is used for clusters which are known to corrupt data due to a flaw in the storage medium, such us a bad memory cell.

6.1.3 Root Directory

The root directory has the same structure as any other directories, which reside in clusters the same way like ordinary files. The difference is that the root directory is allocated when the disk is formatted and it has a fixed and known position and size. Sub-directories are stored on the disk in a similar way to regular files, therefore they can span multiple sectors and their file count can be much larger than that of the root directory.

A directory is organized in 32-byte entries representing individual files. Table 6.2 shows the structure of one such entry.

The name and extension fields form together the well-known 8.3 filename format. Longer file names are encoded using a *long file name* (LFN) scheme as special hidden entries stored in the directory table alongside the regular 8.3 entries, ensuring backward compatibility.

The first byte of the file name has a special meaning:

Offset	Size (bytes)	Description
0	8	File name (padded with spaces)
8	3	File extension
11	1	File attributes
12	10	Reserved
22	2	Creation time
24	2	Creation date
26	2	Address of the first cluster
28	4	File size (bytes)

Table 6.2: Structure of a FAT16 directory entry

- 0x00 - indicates that there are no more files when searching the directory
- 0xE5 - marks a free slot; this is used when a file is deleted
- 0x05 - indicates that the first byte should actually be 0xE5, which was used in a Japanese character set.
- Any other value, except 0x20 (space) and characters forbidden in a DOS file name, starts a valid file entry. Generally, only space, A-Z, 0-9, - and _ should be used in file names for maximum compatibility.

The attributes field contains flags such as *directory*, *volume label*, *read-only* and *hidden*. Volume label is a special entry in the root directory, which defines the disk's label shown on the host PC. A file with the directory bit set is actually a pointer to a subdirectory, meaning that when we open the linked cluster, we'll find a new directory table.

Figure 6.1 shows a possible organization of the GEX file system with two INI files, one spanning two clusters, the other being entirely inside one. The clusters need not be used completely; an exact file size is stored in the directory entries.

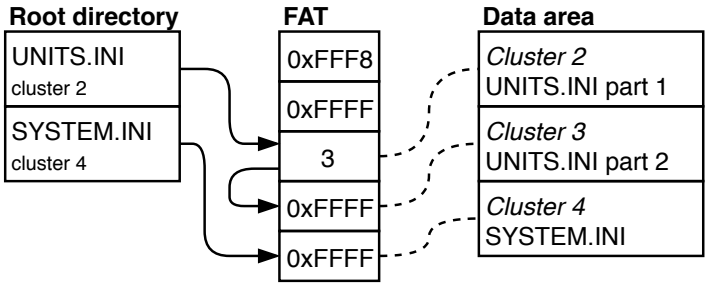


Figure 6.1: An example of the GEX virtual file system

6.2 FAT16 Emulation

reference

The FAT16 file system is relatively straightforward and easy to implement. This is the reason why an emulation driver for it was developed as part of the open-source ARM mbed DAPLink project. It is used there for a drag-and-drop flashing of firmware images to the target microcontroller, taking advantage of it working well across different host platforms. ARM mbed uses a browser-based IDE and cloud build servers, thus the end user does not need to install or set up any software or drivers to program a compatible development kit. The GEX firmware adapts several parts of this code, optimizes its RAM usage and further expands its functionality to support our specific use case.

It is not practical or even possible to keep the entire file system in memory, especially with a microcontroller like the STM32F072, which has only 16 kB of RAM in total. This means that we have to generate and parse disk sectors and clusters on-demand, when the host reads or writes them. The STM32 USB Device library helpfully implements the Mass Storage USB class and provides API endpoints to which we connect our file system emulator. Specifically, those are requests to read and write a sector, and to read disk status and parameters, such as its size.

As shown in table 6.1, the disk consists of several areas. The boot sector is immutable and can be stored in Flash. The handling of the other areas (FAT, data area) depends on whether we're dealing with a read or write access:

6.2.1 Handling a Read Access

The user can only read files that already exist on the disk, in our case, `UNITS.INI` and `SYSTEM.INI`. Those files are generated from the binary settings storage, and conversely, parsed, line-by-line, without ever existing in their full form. This fact makes our task more challenging, as the files can't be easily measured and there's no obvious way to read a sector from the middle of a longer file. We solve this by implementing two additional functions in the INI file writer: a *read window* and a *dummy read mode*.

A read window is a byte range which we wish to generate. The INI writer discards bytes before the start of the read window, writes those inside the window to our data buffer, and stops when its end is reached. This lets us extract a sector from anywhere in a file. The second function, dummy read, is tied to the window function: we set the start index so high that it's never reached (e.g. 0xFFFFFFFF), and have the writer count discarded characters. When the dummy file generation ends, this character counter holds its size.

Now, just one problem remains: how to tell which sectors contain which part of our files? This is straightforward when we realize that the files change only when the user modifies the settings. After each such change, an algorithm is run which allocates clusters to the files and preserves this information in a holding structure. A subsequent read access simply requires a look into this structure and the corresponding chunk of a file may be served using the read window function. The FAT can be dynamically generated from this information as well.

6.2.2 Handling a Write Access

A file write access is more challenging to emulate than a read access, as the host OS tends to be somewhat unpredictable. In GEX's case we're interested only in the action of overwriting an already existing file, but it's interesting to also analyze other actions the host may perform.

It must be noted that due to the nonexistence of a physical storage medium, it's not possible to read back a file the host has written. The OS may show the written file on the disk, but when the user tried to read it, it either fails, or shows a cached copy. In the DAPLink emulator this is worked around by temporarily reporting that the storage medium has been removed, forcing the host to re-load its contents. In GEX, the loaded INI file will be a newly generated copy, embedding possible error messages as comments.

File Deletion

A file is deleted by:

1. Marking all sectors used by it as free in the FAT
2. Replacing the first character of its name in the directory table by 0xE5 to indicate the slot is free

From the perspective of emulation, we can ignore the FAT access and only detect writes to the directory sectors. This is slightly more complicated when one considers that all disk access is performed in sectors: the emulator must compare the written data with the original bytes to detect what change has been performed. Alternatively, we could parse the written sector as a directory table and compare it with our knowledge of its original contents.

6.2.3 File Name Change

A file is renamed by modifying its directory entry. This can be be detected in a similar way to a file deletion. In the simple case of a short, 8.3 file name, this is a in-place modification of the file entry. Long file names, using the LFN extension, are a complication, as the number of dummy entries might change when the file name is shortened or made longer, and subsequently the following entries in the table may shift or be entirely re-arranged.

6.2.4 File Creation

A new file is created in three steps:

1. Finding free clusters and chaining them by writing the following cluster addresses (or 0xFFFF for the last cluster) into the FAT
2. Finding and overwriting a free entry in the directory table

3. Writing the file content

It can be expected the host OS first finds the free sectors and a free file entry before performing any write operations, to prevent a potential disk corruption.

To properly handle such a file by the emulator, we could, in theory, find its name from the directory table, which has been updated, and then collect the data written to the corresponding clusters. In practice, confirmed by experiments with a real Linux host, those three steps may happen in any order, and often the content is written before the directory table is updated.

The uncertain order of the written disk areas poses a problem when the file name has any significance, as we can't store the received file data while waiting for the name to be written. The DAPLink mbed flashing firmware solves this by analyzing the content of the first written sector of the file, which may contain the binary NVIC table, or an ASCII pattern typical for Intel hex files.

6.2.5 File Content Change

A change to a file's content is performed in a similar way to the creation of a new file, except instead of creating a new entry in the directory table, an existing one is updated with the new file size. The name of the file may be, again, unknown until the content is written, but we could detect the file by comparing the starting sector with those of all files known to the virtual file system.

In the case of GEX, the detection of a file name is not important; We expect only INI files to be written, and the particular file may be detected by its first section marker, such as `[UNITS]` or `[SYSTEM]`. Should a non-INI file be written by accident, the INI parser will likely detect a syntax error and discard it.

It should be noted that a file could be updated only partially, skipping the clusters which remain unchanged, and there's also no guarantee regarding the order in which the file's sectors are written. This is hard to detect and handle and the current firmware is not able to interpret it correctly, thus such a write operation will fail. Fortunately, this host behavior has not been conclusively observed in practice, but a write operation rarely fails for still unknown reasons and this could be a possible cause.

Chapter 7

Supported Hardware Buses

Hardware buses implemented in GEX are presented in this chapter. The description of each bus is accompanied by several examples of devices that can be interfaced with it. The reader is advised to consult the official specifications and particular devices' datasheets for additional technical details.

7.1 UART and USART

The *Universal Synchronous / Asynchronous Receiver Transmitter* has a long history and is still in widespread use today. It is the protocol used in RS-232, which was once a common way of connecting modems, printers, mice and other devices to personal computers. UART framing is also used in the industrial bus RS-485.

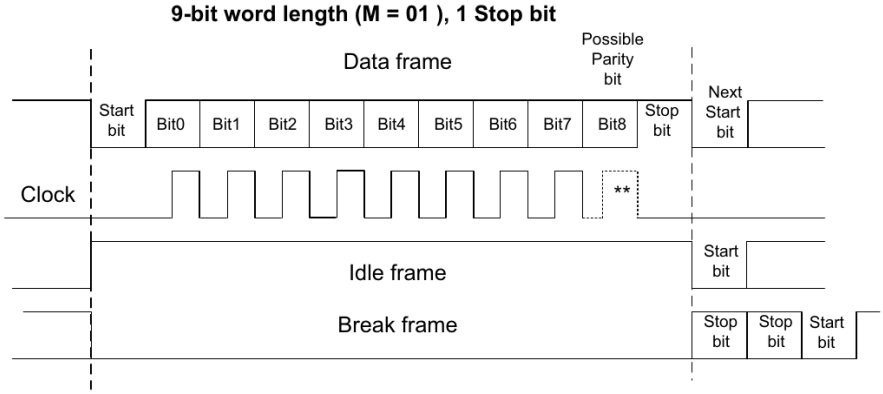


Figure 7.1: UART frame, as shown by the STM32F072 Reference Manual. Break frames are used by some UART based protocols, like LIN (Local Interconnect Network).

UART and USART are two variants of the same interface. USART includes a separate clock signal, while the UART timing relies on a well-known clock speed and the bit clock is synchronized by start bits. USART was historically used in modems to achieve higher bandwidth, but is now mostly obsolete.

USART, as implemented by microcontrollers such as the STM32 family, is a two-wire full duplex interface that uses 3.3 V or 5 V logic levels. The data lines are in the high logical level when idle. A frame, pictured in figure 7.1 starts by a start-bit (low level for the period

of one bit) followed by n data bits (typically eight), an optional parity bit and a period of high level called a stop bit or stop bits, usually between one and two bits long.

RS-232 uses the UART framing, but its logic levels are different: logical 1 is represented by negative voltages -3 to $-25\,\mathrm{V}$ and logical 0 uses the same range, but positive. To convert between RS232 levels and TTL ($5\,\mathrm{V}$) levels, a level-shifting circuit such as the MAX232 can be used. In RS232, the two data lines (Rx and Tx) are accompanied by RTS (Ready To Send), CTS (Clear To Send) and DTR (Data Terminal Ready) which facilitate handshaking and hardware flow control. In practice, those additional signals are often unused or their function differs; for instance, Arduino boards (using a USB-serial converter) use the DTR line as a reset signal to automatically enter their bootloader for firmware flashing.

7.1.1 Examples of Devices Using UART

- **MH-Z19B** - NDIR CO2 concentration sensor
- **NEO-M8** - uBlox GPS module
- **ESP8266** with AT firmware - a WiFi module
- **MFRC522** - NFC MIFARE reader/writer IC (also supports other interfaces)

7.2 SPI

SPI (Serial Peripheral Interface) is a point-to-point or multi-drop master-slave interface based on shift registers. The SPI connection with multiple slave devices is depicted in figure 7.2. It uses at least 4 wires: SCK (Serial Clock), MOSI (Master Out Slave In), MISO (Master In Slave Out) and SS (Slave Select). SS is often marked CSB (Chip Select Bar) or NSS (Negated Slave Select) to indicate it's active low. Slave devices are addressed using their Slave Select input while the data connections are shared. A slave that's not addressed releases the MISO line to a high impedance state so it doesn't interfere in ongoing communication.

Transmission and reception on the SPI bus happen simultaneously. A bus master asserts the SS pin of a slave it wishes to address and then sends data on the MOSI line while receiving a response on MISO. It's customary that the slave responds with zeros or a status byte as the first byte of the response.

SPI devices often provide a number of control, configuration and status registers that can be read and written by the bus master. The first byte of a command usually contains one bit that determines if it's a read or write access, and an address field selecting the target register.

7.2.1 Examples of Devices Using SPI

- **SX1276** - LoRa transceiver

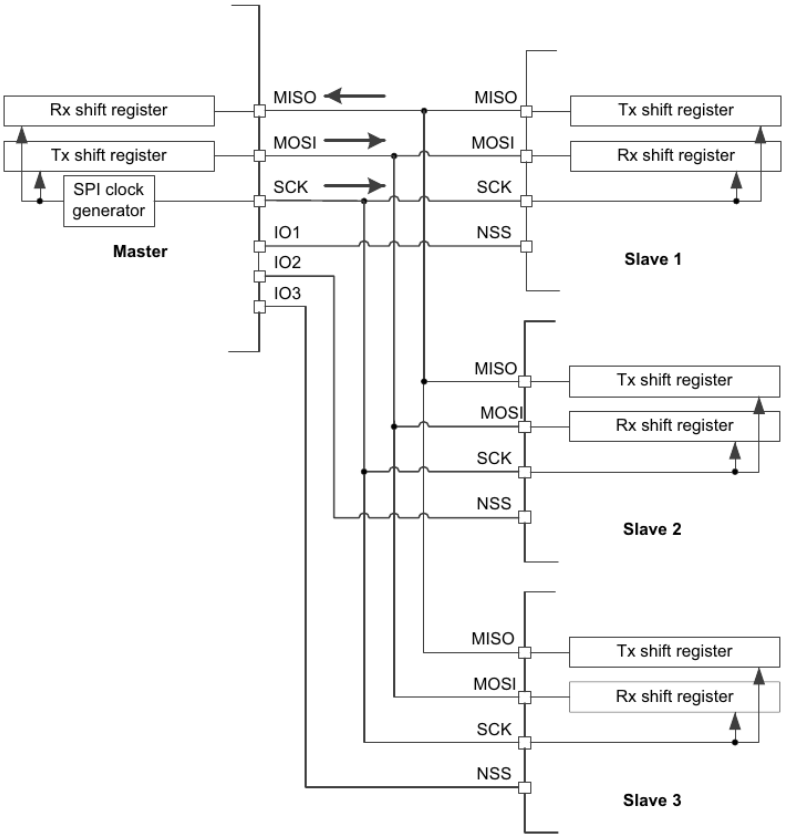


Figure 7.2: A SPI bus with 1 master and 3 slaves, each enabled by its own Slave Select signal (*STM32F072 Reference Manual*)

- **nRF24L01+** - 2.4 GHz ISM band radio module
- **L3GD20** - 3-axis gyroscope
- **BMP280** - pressure sensor
- **BME680** - air quality sensor
- **ENC28J60** - Ethernet controller
- **L6470** - intelligent stepper motor driver
- **AD9833** - DDS-based DAC / waveform generator (MOSI only)
- **ADE7912** - triple Σ-Δ ADC for power metering applications
- **SD cards**
- SPI-interfaced EEPROMs and Flash memories

7.3 I2C

I2C is a two-wire (SDA–*Serial Data*, SCL–*Serial Clock*), open-drain bus that supports multi-master operation. The protocol was developed by Philips Semiconductor (now NXP Semiconductors) and until 2006 implementors were required to pay licensing fees, leading to the development of compatible implementations with different names, such as Atmel's Two Wire Interface (TWI) or Dallas Semiconductor's "Serial 2-wire Interface" (e.g. used in the DS1307 RTC chip). I2C is the basis of the SMBus and PMBus protocols which add additional constraints and rules for a more robust operation.

I2C uses two addressing modes: 7-bit and 10-bit. Due to the small address space, exacerbated by many devices implementing only the 7-bit addressing, collisions between chips from different manufacturers are common; many devices thus offer several pins to let the board designer choose a few bits of the address by connecting them to different logic levels. I2C allows slow slave devices to stop the master from sending more data by holding the SCL line low at the end of a byte. As the bus is open-drain, the line can't go high until all participants release it. This function is called *Clock Stretching*.

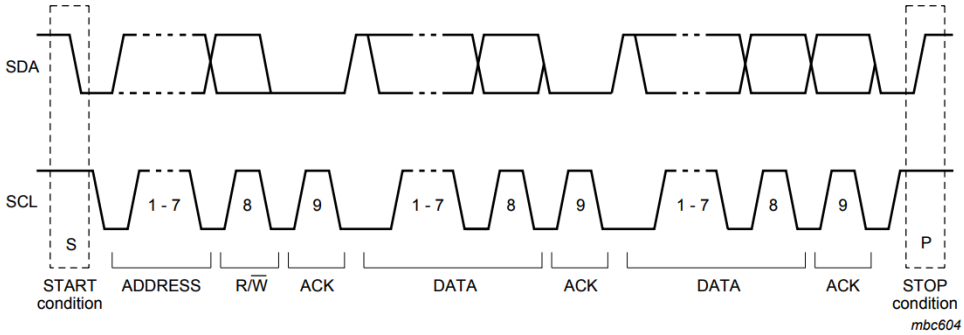


Figure 7.3: An I2C message diagram. The frame starts with a start condition and stops with a stop condition, defined by an SDA edge while SCL is high. The address and data bytes are acknowledged by the slave by sending a 0 on the open-drain SDA line in the following clock cycle. A slave can terminate the transaction by sending 1 in place of the acknowledge bit. (*Diagram taken from the I2C specification UM10204 by NXP Semiconductors*)

The bus supports multi-master operation, which leads to the problem of collisions. Multi-master capable devices must implement a bus arbitration scheme as specified by the I2C standard. This feature is not often used in intelligent sensors and modules; the most common topology is multi-drop single-master, similar to SPI, with the advantage of using only two pins on the microcontroller.

7.3.1 Examples of Devices Using I2C

- **APDS-9960** - ambient light, proximity and gesture sensor
- **L3GD20**, **BMP280**, **BME680** - listed as SPI devices, those also support I2C

- **DS1307** - RTC; I2C is not mentioned in the entire datasheet, presumably to avoid paying license fees, but it is fully compatible
- **IS31FL3730** - LED matrix driver
- Cameras with an SCCB port can be accessed through I2C

7.4 1-Wire

The 1-Wire bus, developed by Dallas Semiconductor, uses a single bi-directional data line which can also power the slave devices, reducing the number of required wires to just two (compare with 3 in I2C and 5 in SPI, all including GND).

1-Wire is open-drain and the communication consists of short pulses sent by the master and (for bit reading) the line continuing to be held low by the slave. The pulse timing (fig. 7.5) defines if it's a read or write operation and what bit value it carries. A transaction is started by a 480us long "reset" pulse send by master and ended by a 1-byte CRC checksum.

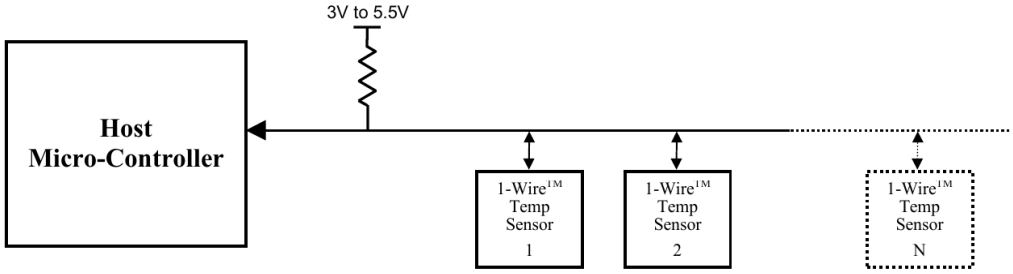


Figure 7.4: 1-Wire topology (by *Dallas Semiconductor*)

1-Wire is a master-slave multi-drop bus. Devices are addressed by their unique 64-bit ID numbers (called ROMs); those IDs are found by the bus master with the cooperation from slaves using a ROM search protocol. If only one device is connected, a special command set can be used to skip addressing.

7.4.1 Examples of Devices Using 1-Wire

- **DS1820**, **DS18S20**, **DS18B20** - digital thermometers
- **iButton** - contact-read access tokens, temperature loggers etc.

Since 1-Wire is a proprietary protocol, there is a much smaller choice of available devices and they also tend to be more expensive. The DS18x20 thermometers are, however, popular enough to warrant the bus's inclusion in GEX.

Explain the ROM Search algorithm

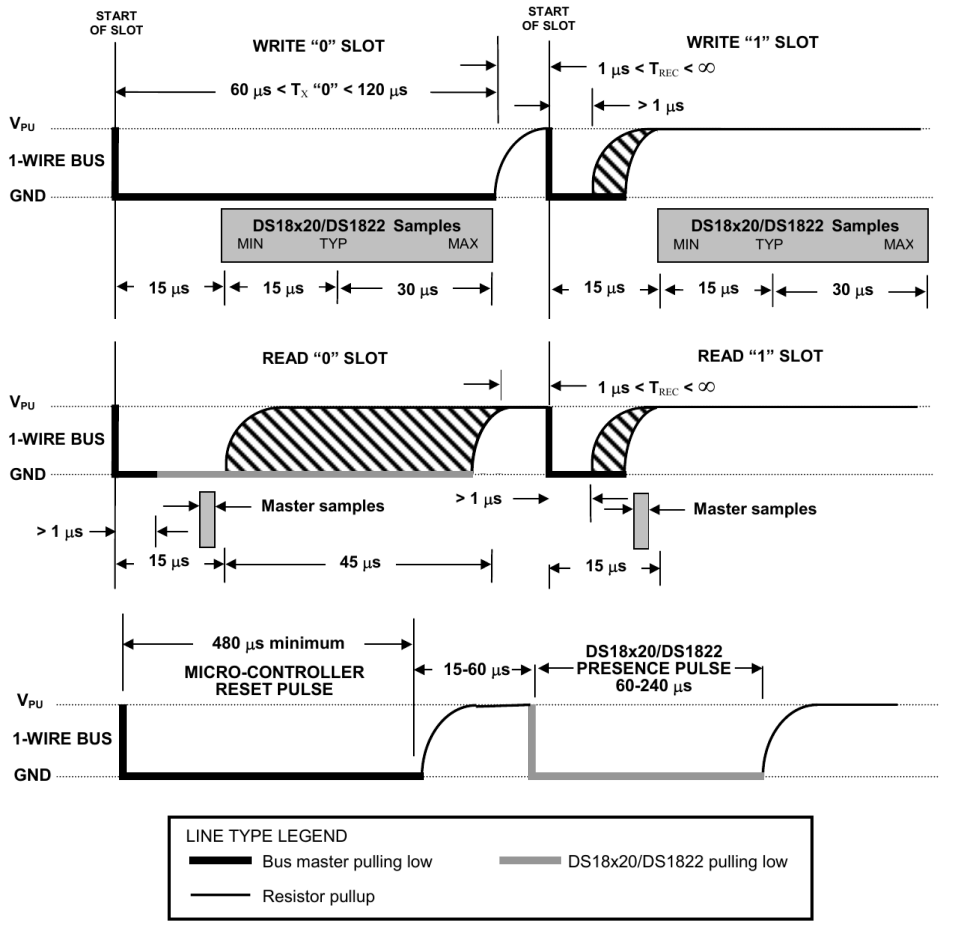


Figure 7.5: The 1-Wire DIO pulse timing (by *Dallas Semiconductor*)

7.5 NeoPixel

NeoPixel is a marketing name of the **WS2811**, **WS2812** and compatible intelligent LED drivers that is commonly used in "addressable LED strips". Those chips include the control logic, PWM drivers and usually the LED diodes all in one miniature package.

The NeoPixel protocol is unidirectional, using only one data pin. The LED drivers are chained together. Ones and zeros are encoded by a pulse length on the data pin; after loading the color data to the LED string, a longer "reset" pulse is issued by the bus master and the set colors are displayed. The timing diagram and constraints are shown in table 7.1.

The NeoPixel timing is very sensitive to pulse length accuracy. Reliable ways to implement it use DMA with a hardware timer, or a I2S peripheral. An easier method that does not use any additional hardware resources is implementing the protocol as delay loops in the firmware; care must be taken to disable interrupts in the sensitive parts of the timing.

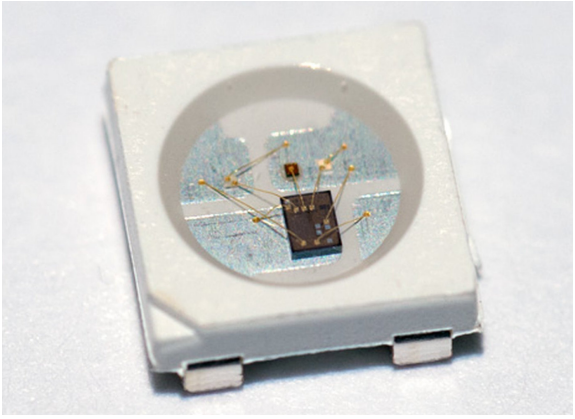


Figure 7.6: A close-up photo of the WS2812B package, showing the LED driver IC

Bit value	Constraint	Duration
0	High level	$0.4\,\mu s \pm 150ns$
0	Low level	$0.85\,\mu s \pm 150ns$
1	High level	$0.45\,\mu s \pm 150ns$
1	Low level	$0.8\,\mu s \pm 150ns$
–	Reset pulse (low)	$> 50\,\mu s$

Table 7.1: NeoPixel pulse timing, according to the datasheet

Chapter 8

Non-communication Hardware Functions

In addition to communication buses, described in chapter 7, GEX implements several measurement and output functions that take advantage of the microcontroller's peripheral blocks, such as timers/counters and DAC. The more complicated ones are described here; simpler functions, such as the raw GPIO access, will be described later together with their control API.

8.1 Frequency Measurement

Applications like motor speed measurement and the reading of a VCO (voltage-controlled-oscillator) or VCO-based sensor's output demand a tool capable of measuring frequency. This can be done using a laboratory instrument such as the Agilent 53131A. A low cost solution is to use a timer/counter peripheral of a microcontroller, such as the STM32F072 used in GEX.

Two basic methods to measure frequency exist, each with it's advantages and drawbacks:

- The *direct method* (fig. 8.1) is based on the definition of frequency as a number of cycles n in a fixed-length time window τ (usually 1 s); the frequency is then calculated as $f = n/\tau$.

  One timer generates the time window and its output gates the input of another, configured as a pulse counter. At the end of the measurement window an interrupt is generated and we can read the pulse count from the counter's register.

  The direct method has a resolution of 1 Hz with a sampling window of 1 s (only a whole number of pulses can me counted). The resolution can be increased by using a longer time window, provided the measured signal is stable enough to make averaging possible without distorting the result.

- The *indirect* or *reciprocal method* (fig. 8.2) measures one period T as the time interval between two pulses and this is then converted to frequency as $f = 1/T$.

  This method needs only one timer/counter. Cycles of the system clock are counted for the duration of one period on the input pin (between two rising edges). If we additionally detect the falling edge in between, the counter's value gives us the duty cycle when related to the overall period length.

  Te reciprocal method's resolution depends on the counter's clock speed; if driven at 48 MHz, the tick period is 20.83 ns, which defines the granularity of our time

measurement. It is common to measure several pulses and average the obtained values to further increase the precision.

We can easily achieve a sub-hertz resolution with this method, but its performance degrades at high frequencies where the time measurement precision becomes insufficient. The input frequency range can be extended using a hardware prescaller[1], which is also applicable to the direct method, should the measurement of frequencies outside the counter's supported range be required. A duty cycle measurement available in this method can be used to read the output of sensors that use a pulse-width modulation.

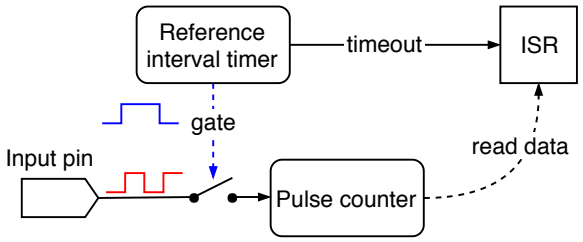


Figure 8.1: Direct frequency measurement method

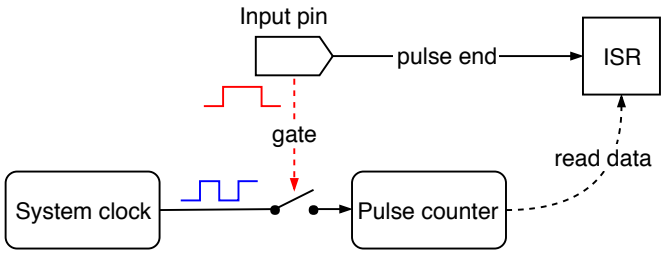


Figure 8.2: Reciprocal frequency measurement method

Which method to use depends on the frequency we want to measure; the worst-case measurement errors of both methods, assuming an ideal 48 MHz system clock, are plotted in figure 8.3. It can be seen that the reciprocal method leads in performance up to 7 kHz where the direct method overtakes it. If a higher error is acceptable, the reciprocal method could be used also for higher frequencies to avoid a reconfiguration and to take advantage of its higher speed.

A good approach to a universal measurement, when we don't know the expected frequency beforehand, could be to first obtain an estimate using the direct method, and if the frequency is below the worst-case error crossing point (here 7 kHz), to take a more precise measurement using the reciprocal method.

The system clock's frequency, which we use to measure pulse lengths and to gate the pulse counter, will be affected by tolerances of the used components, the layout of the PCB, temperature effects etc., causing measurement errors. A higher accuracy could be achieved using a temperature-compensated oscillator (TCO), or, in the direct method, by using the synchronization pulse provided by a GPS receiver to time the measurement interval.

[1] *Prescaller* is a divider implemented as part of the timer/counter peripheral block that can be optionally enabled and configured to a desired division factor.

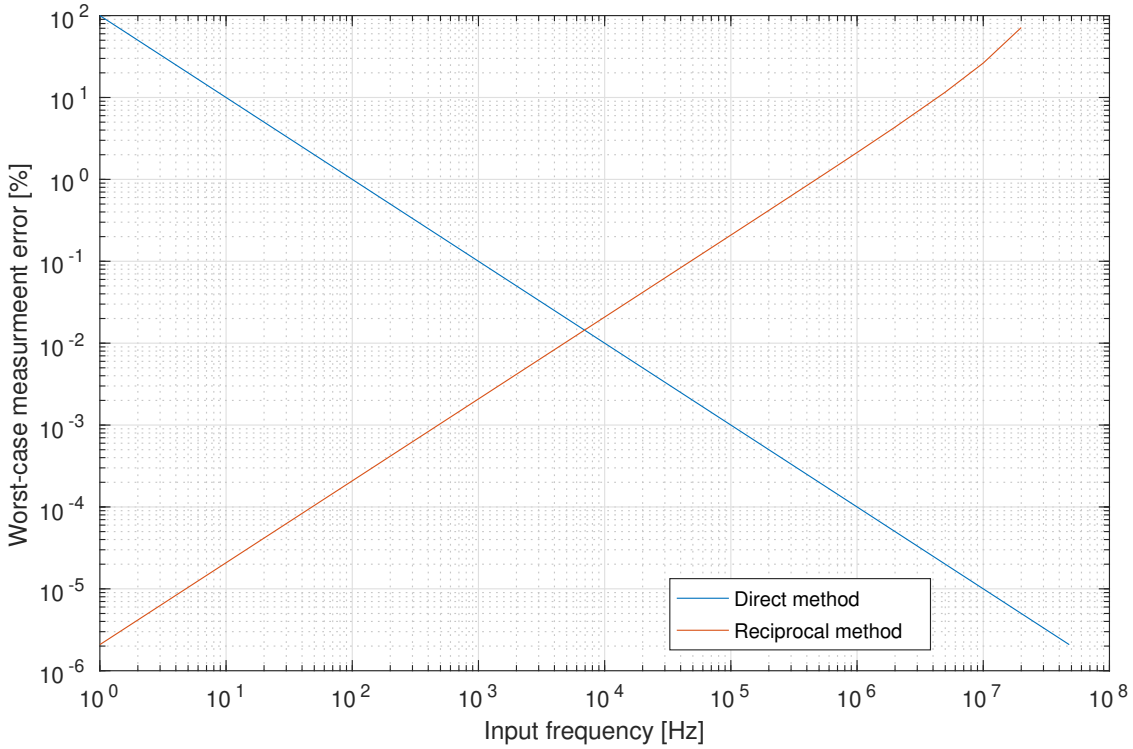


Figure 8.3: Worst-case error using the two frequency measurement methods with an ideal 48 MHz timer clock. The crossing lies at 7 kHz with an error of 0.015 %, or 1.05 Hz.

8.2 Analog Signal Acquisition

A very common need in experiments involving the measurement of physical properties is the acquisition of analog signals, respective voltages. Those can be roughly divided into DC and time-changing signals. Analog signals are converted to digital values using ADCs (Analog to Digital Converters). Several principles of analog signal measurement exist with different cost, speed, resolution, and many other factors which determine their suitability for a particular application.

DC signals can be measured by taking several samples and calculating their average value; In the presence of a 50 Hz or 60 Hz mains interference, its advisable to spread those samples over the 20 ms (resp. 16.7 ms) time of one period so that the interfering waveform cancels out. Time-changing signals can be captured by taking isochronous samples at a frequency conforming to the Nyquist theorem, that is, at least twice that of the measured signal. In practice, a frequency several times higher is preferred for a more accurate capture.

The ADC type commonly available in microcontrollers, including our STM32F072, uses a *successive approximation* method (it is often called the SAR type ADC, after its main component, the *successive approximation register*). Its diagram is shown in figure 8.4.

The SAR type converter uses a DAC (Digital to Analog Converter) which approximates the input voltage, bit by bit, starting from the MSB, following the algorithm outlined below.

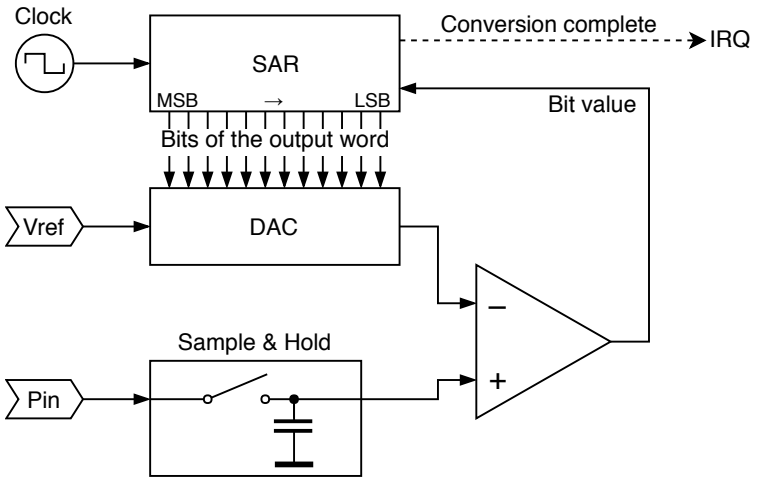


Figure 8.4: A diagram of the SAR type ADC

1. The SAR is cleared to all zeros.
2. The DAC generates an approximation voltage.
3. Its output is compared with the sampled input, and the comparator's output is stored as the active bit in the approximation register.
4. The approximation continues with step 2 and the following (less significant) bit.
5. When all bits of the data word were found, an interrupt request is generated and the application program can read it from the SAR.

A change of the input value would make this principle unreliable, which is why the input is buffered by a sample & hold circuit. The holding capacitor is charged to the input voltage and maintains this level during the conversion. The duration for which the capacitor is connected to the input is called a *sampling time*.

8.3 Waveform Generation

A waveform generator is a useful tool in many experiments and measurements. A sine stimulus is the basis of a lock-in amplifier; it can be used to measure impedance; with a frequency sweep, we can obtain the frequency response of an analog filter, etc. We can, of course, generate other waveforms, such as a triangle, ramp, or rectangle wave.

The DAC peripheral can produce a DC level on the output pin based on a control word. When we periodically change its digital input, it produces an analog waveform.

8.3.1 Waveform Generation with DMA and a Timer

A straightforward implementation of the waveform generator is illustrated in figure 8.5. This approach has its advantages: it's simple and works entirely in the background, with

no interrupt handling required. It could even be implemented entirely in software, using a loop periodically updating the DAC values, of course such approach is less flexible and we would run into problems with asynchronous interrupts.

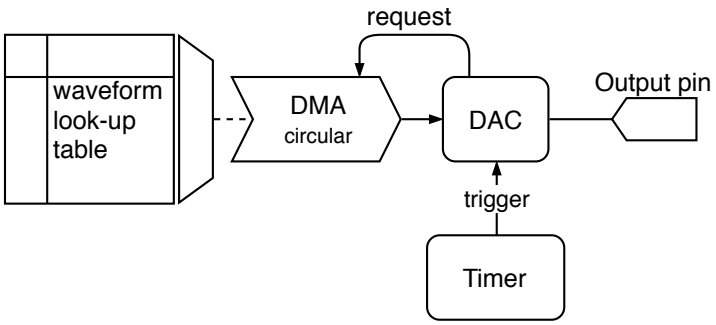


Figure 8.5: A simple implementation of the waveform generator, using DMA and a look-up table

The highest achievable output frequency largely depends on the size of our look-up table. For instance, assuming a timer frequency of 48 MHz and a 8192-word table, holding one period of the waveform, the maximum frequency would be short of 6 kHz, whereas if we shorten the table to just 1024 words, we can get almost 47 kHz on the analog output. The downside of a shorter table is a lower resolution, which will appear as DC plateaus or steps when observed with an oscilloscope, producing harmonic components similar to those of a square wave.

A major disadvantage of this simple generation method is given by the limitations of the used timer, which defines the output frequency. Its output trigger fires when the internal counter reaches a pre-defined value, after which the counting register is reset. The counting speed is derived from the system clock frequency f_c using a prescaller P and the set maximum value N. Only output frequencies that can be exactly expressed as $f = f_c/(P \cdot N \cdot \text{TableSize})$ can be accurately produced. Still, this simple and efficient method may be used where fine tuning is not required to take advantage of its fully asynchronous operation.

8.3.2 Direct Digital Synthesis

There are situations where the simple waveform generation method is not sufficient, particularly when a fine tuning or on-line frequency and phase changes are required. Those are the strengths of a signal generation method called *Direct Digital Synthesis* (DDS).

A diagram of a possible DDS implementation in the STM32 firmware is shown in figure 8.6. It is based on a *numerically controlled oscillator* (NCO). NCO consists of a *phase accumulator* register and a *tuning word* which is periodically added to it at a constant rate in a timer interrupt handler. The value of the tuning word determines the output waveform frequency. The look-up table must have a power-of-two length so that it can be addressed by the n most significant bits of the phase accumulator. An additional control word could be added to this address to implement a phase offset for applications like a phase-shift modulation.

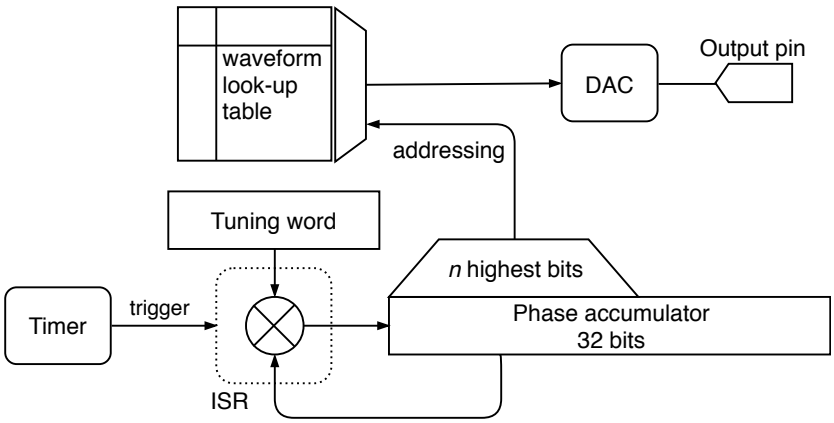


Figure 8.6: A block diagram of a DDS-based waveform generator

The output frequency is calculated as $f_{\text{out}} = \frac{M \cdot f_{\text{c}}}{2^n}$, where M is the tuning word, n is the bit length of the phase accumulator, and f_c is the frequency of the phase-updating interrupt. The number of bits used to address the look-up table does not affect the output frequency; the table can be as large as the storage space allows. A tuning word value exceeding the lower part of the phase accumulator (including bits which directly enter the look-up address) will cause some values from the table to be skipped. A smaller tuning word, conversely, makes some values appear on the output more than once. This can be observed as steps or flat areas on the output. When the tuning word does not evenly divide 2^n, that is, the modulo is non-zero, we can also observe jitter.

DDS Implemented in Hardware

DDS may be implemented in hardware, including the look-up table, often together with the DAC itself, which is then called a *Complete DDS*. That is the case of e.g. AD9833 from Analog Devices. As the software implementation depends on a periodic interrupt, it's often advantageous to use a component like this when we need higher output frequencies where the use of an interrupt is not possible. GEX can control an external waveform generator like the AD9833 using an SPI port.

8.4 Touch Sensing

The used microcontroller, STM32F072, includes a touch sensing controller (TSC) peripheral block. It can be accessed from GEX as a demonstration of capacitive touch sensing, and could possibly be used for simple touch sensors as well, such as measuring the level of water in a tank.

The TSC requires a specific topology with a sampling capacitor connected close to the microcontroller pin, which may not be possible on a universal GEX module; for this reason, the touch sensing feature is best demonstrated on the STM32F072 Discovery development kit, which includes a 4-segment touch slider shown in figure 8.7.

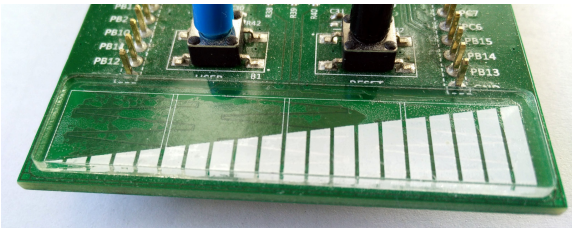


Figure 8.7: The touch slider on a STM32F072 Discovery board

The principle of capacitive touch sensing is well explained in the microcontroller's reference manual XXX. A key part of the TSC is a set of analog switches, which can be combined to form several different signal paths between the external pins, Vdd, GND, and an analog comparator. Two input pins are needed for every touch sensing channel: the sensing pad connects to one, the other is connected through a sampling capacitor (47 nF on the Discovery board) to GND.

ref

Capacitive sensing is a sequential process described in the following steps:

1. The sensing capacitor is discharged by connecting its free end to GND.
2. The sensing pad is connected to Vdd and, acting as a capacitor, charged to 3.3 V. It stores a small amount of charge, depending on its capacitance—this is the variable property we are trying to measure.
3. The free terminals of the two capacitors (the sensing pad and the sampling capacitor) are connected together and their voltages reach an equilibrium as a portion of the stored charge leaves the sensing pad and flows into the bigger capacitor.
4. The steps (2) and (3) are repeated the sampling capacitor's voltage exceeds a fixed threshold (set to a half of the supply voltage). The number of cycles needed to charge the sampling capacitor corresponds to the capacitance of the sensing pad.

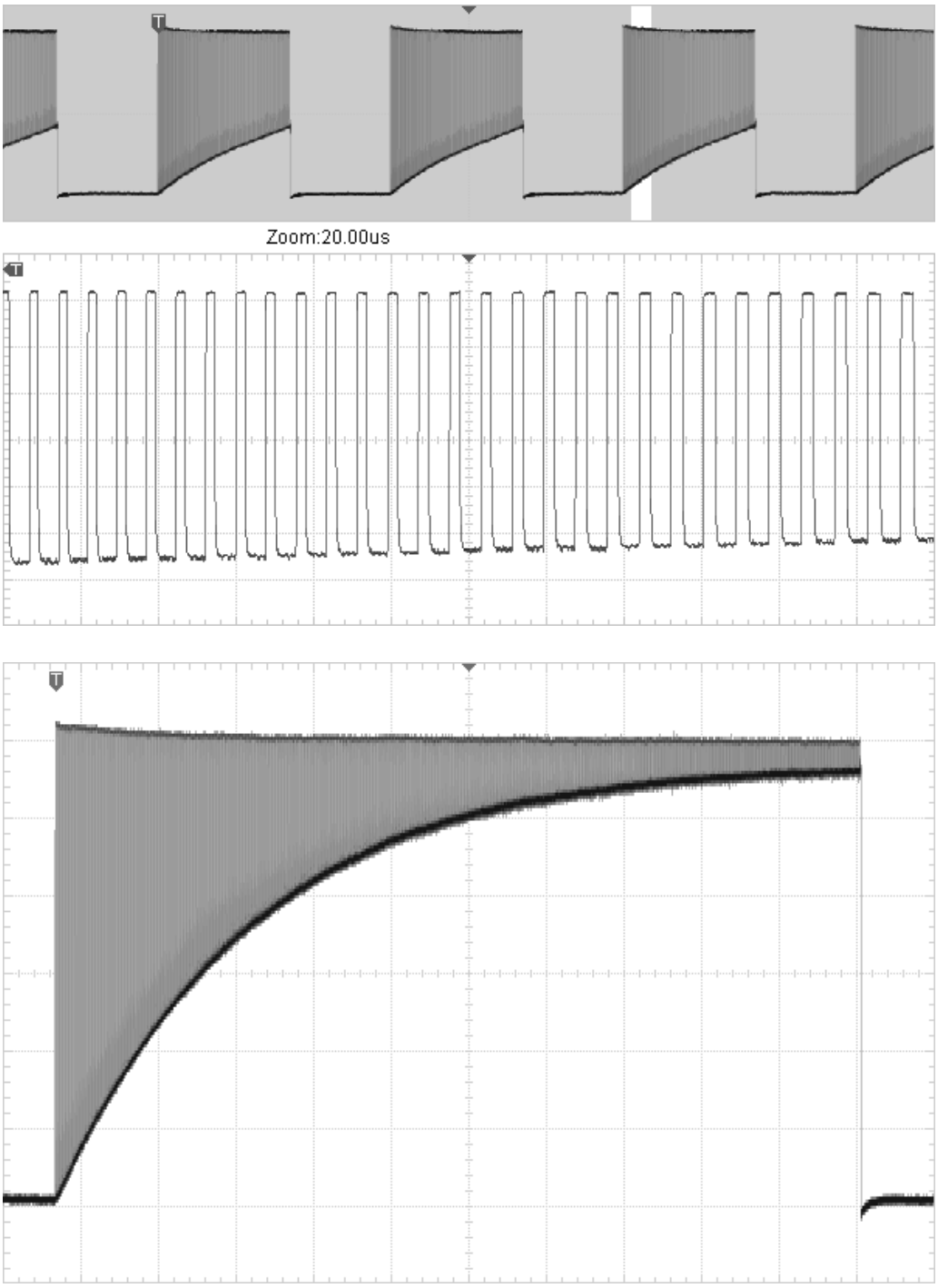


Figure 8.8: A voltage waveform measured on the touch sensing pad. The bottom side of the envelope equals the sampling capacitor's voltage—this is the phase where both capacitors are connected. The detailed view (middle) shows the individual charging cycles. The bottom screenshot captures the entire waveform after the analog comparator was disabled.

Part III

Implementation

Chapter 9

Application Structure

GEX is designed to be modular and easy to extend. It's composed of a set of functional blocks, sometimes available in more than one instance, which can be configured by the user to fit their application needs. The firmware is built around a *core framework* which provides services to the functional blocks, such as a settings storage, resource allocation, message delivery and periodic updates.

In this chapter, we will focus on the general function of the GEX module and will look at the services provided by the core framework. Individual functional blocks and the control API will be described in the following chapters. references

A writing style note: This and the following parts were written after implementing and evaluating the first hardware prototype and its firmware, therefore rather than describing the development process, it tends to talk about the completed solution and the decisions taken.

9.1 User's View of GEX

Before going into implementation details, we'll have a look at GEX from the outside, how an end user will see it. This should give the reader some context to better orient themselves in the following sections and chapters investigating the internal structure of the firmware and the communication protocol.

The GEX firmware can be flashed to a STM32 Nucleo or Discovery board or a custom PCB. It's equipped with a USB connector to connect to the host PC. GEX loads its configuration from the non-volatile memory, configures its peripherals, sets up the function blocks and enables the selected communication interface(s). When USB is connected to the board, the PC enumerates it and either recognizes the communication interface as CDC/ACM (Virtual serial port), or leaves it without a software driver attached, to be accessed directly as raw USB endpoints. This can be configured. The user can now access the functional blocks using the client library and the serial protocol, as well as modify the configuration files.

The board is equipped with a button or a jumper labeled LOCK. When the button is pressed or the jumper removed, the Mass Storage USB interface is enabled. For the user this means a new disk will be detected by their PC's operating system that they can open in a file manager. This disk provides read and write access to configuration INI files and other files with useful information, like a list of supported features and available hardware

resources. The user now edits a configuration file and saves it back to the disk. GEX processes the new content, tries to apply the changes and generates an updated version of the file that includes error messages if there was a problem. For the PC OS to recognize this change, the Mass Storage device momentarily reports that the media is unavailable to force the OS to reload it. This is a similar mechanism to what happens when a memory card is removed from a reader. Now the user must reload the file in their editor, inspect the updated content and perform any changes needed. The settings, when applied successfully, should now be available to test using the communication interface. When everything is to the user's satisfaction, the updated settings are committed to the device's non-volatile memory by pressing the LOCK button again, or replacing the jumper.

For boards without a USB re-enumeration capability (notably with older microcontrollers like the STM32F103) that use a jumper, this must be removed before plugging the board to the host USB so that the Mass Storage is enabled immediately at start-up and a re-enumeration is not needed.

In the case when a wireless communication module is installed on the PCB and GEX is configured to use it, this will be used as a fallback when the USB peripheral does not receive an address (get enumerated) within a short time after start-up. The wireless link works in the same way as any other communication interface: it can be used to read and modify the configuration files and to access the functional blocks. To use it, the user needs to connect a wireless gateway module to their host PC and use the radio link instead of a USB cable. The gateway could support more than once GEX board at once.

Now that GEX is connected and configured, the user can start using it. This involves writing a program in C or Python that uses the GEX client library, using the Python library from MATLAB, or controlling GEX using a GUI front-end built on those libraries. The configuration can be stored in the module, but it's also possible to temporarily (or permanently) replace it using the communication API. This way the settings can be loaded automatically when the user's program starts.

9.2 Functions of the Core Framework

The core framework forms the skeleton of the firmware and usually doesn't need any changes when new user-facing features are added. It provides the following services:

- Hardware resource allocation (9.3)
- Settings storage and loading (9.4)
- Functional block (*units*) initialization (9.5)
- The communication port with different back-ends: USB, UART, wireless (9.7)
- Message sending and delivery (9.8)
- Interrupt management and routing to functional blocks (9.9)
- Virtual mass storage for configuration file editing

When the firmware needs to be ported to a different STM32 microcontroller, the core framework is relatively straightforward to adapt and the whole process can be accomplished in a few hours. The time consuming part is modifying the functional blocks to work correctly with the new device's hardware.

9.3 Resource Allocation

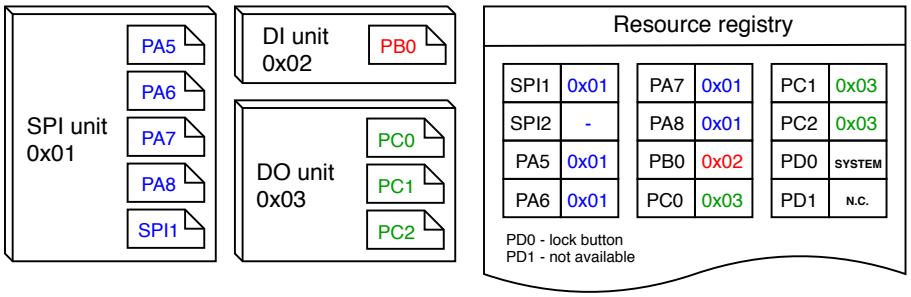


Figure 9.1: An example allocation in the resource registry

The microcontroller provides a number of hardware resources that require exclusive access: GPIO pins, peripheral blocks (SPI, I2C, UART...), DMA channels. If two units tried to control the same pin, the results would be unpredictable; similarly, with a multiple access to a serial port, the output would be a mix of the data streams and completely useless.

To prevent a multiple access, the firmware includes a *resource registry* (fig. 9.1). Each individual resource is represented by a field in a resource table together with its owner's callsign. Initially all resources are free, except for those not available on the particular platform (i.e. a GPIO pin PD1 may be disabled if not present on the microcontroller's package).

The resources used by the core framework are taken by a virtual unit `SYSTEM` on start-up to prevent conflicts with the user's units. This is the case of the status LED, the LOCK button, USB pins, the communication UART, the pins and an SPI peripheral connecting the wireless module, pins used for the crystal oscillator, and the timer/counter which provides the system timebase.

9.4 Settings Storage

The system and unit settings are written, in a binary form, into designated pages of the microcontroller's Flash memory. The unit settings serialization and parsing is implemented by the respective unit drivers.

As the settings persist after a firmware update, it's important to maintain backwards compatibility. This is achieved by prefixing the unit's settings by a version number. When the settings are loaded by a new version of the firmware, it first checks the version and

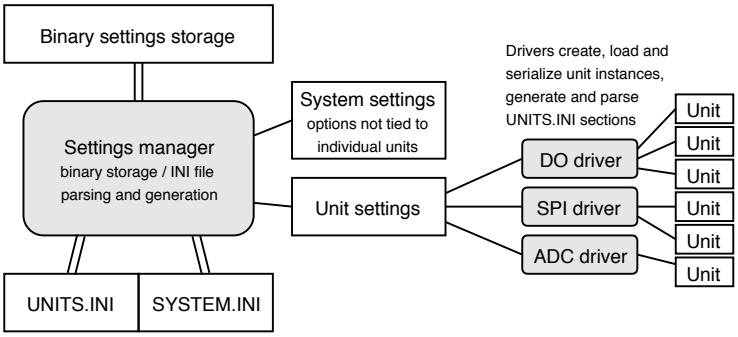


Figure 9.2: Structure of the settings subsystem

decides whether to use the old or new format. When the settings are next changed, the new format will be used.

The INI files, which can be edited through the communication API or using a text editor with the virtual mass storage, are parsed and generated on demand and are never stored in the Flash or RAM, other than in short temporary buffers. The INI parser processes the byte stream on-the-fly as it is received, and a similar method is used to build a INI file from the configured units and system settings.

9.5 Functional Blocks

GEX's user-facing functions, also called functional blocks or *units*, are implemented in *unit drivers*. Those are independent modules in the firmware that the user can enable and configure using the GEX configuration files. In principle, there can be multiple instances of each unit type. However, we are limited by hardware constraints: there may be only one ADC peripheral, two SPI ports and so on. The mutually exclusive assignment of resources to units is handled by the *resource registry* (9.3).

Each unit is defined by a section in the configuration file `UNITS.INI`. It is given a name and a *callsign*, which is a number that serves as an address for message delivery. A unit is internally represented by a data object with the following structure:

- Name
- Callsign
- Configuration parameters loaded from the unit settings
- State variables updated at run-time by user commands or internal functions
- A reference to the unit driver

The unit driver handles commands sent from the host PC, initializes and de-initializes the unit based on its settings, and implements other aspects of the unit's function, such as periodic updates and interrupt handling. Unit drivers may expose public API functions to make it possible to control the unit from a different driver, allowing the creation of "macro units".

9.6 Source Code Layout

Looking at the source code repository (fig. 9.3), at the root we'll find device specific driver libraries and files provided by ST Microelectronics, the FreeRTOS middleware, and a folder called `User` containing the GEX application code. This division is useful when porting the firmware to a different microcontroller, as the GEX folder is mostly platform-independent and can be simply copied (of course, adjustments are needed to accompany different hardware peripheral versions etc.). The GEX core framework consists of everything in the `User` folder, excluding the `units` directory in which the individual units are implemented. Each unit driver must be registered in the file `platform.c` to be available for the user to select. The file `plat_compat.c` includes platform-specific headers and defines e.g. which pin to use for a status LED or the LOCK button.

The USB Device library, which had to be modified to support a composite class, is stored inside the `User` folder too, as it is compatible with all STM32 microcontrollers that support USB.

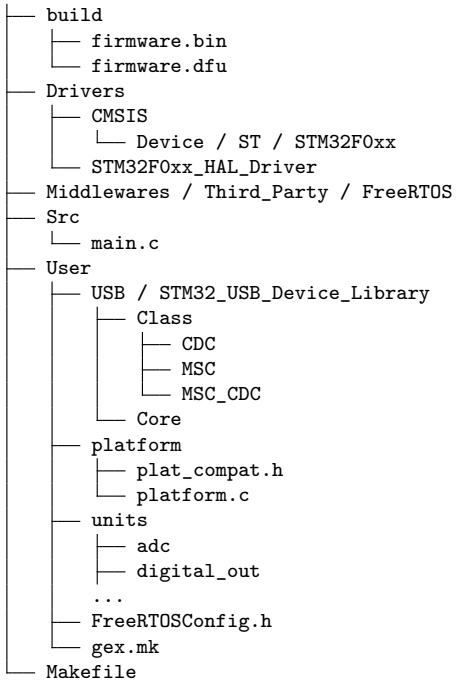


Figure 9.3: The general structure of the source code repository

9.7 Communication Ports

The firmware supports three different communication ports: hardware UART, USB (virtual serial port), and a wireless connection. Each interface is configured and accessed in a different way, but for the rest of the firmware (and for the PC-side application) they all appear as a full duplex serial port. To use interfaces other than USB, the user must configure those in the system settings (a file `SYSTEM.INI` on the configuration disk).

At start-up, the firmware enables the USB peripheral, configures the device library and waits for enumeration by the host PC. When not enumerated, it concludes the USB cable is not connected, and tries some other interface. The UART interface can't be tested as reliably, but it's possible to measure the voltage on the Rx pin. When idle, a UART Rx line should be high (here 3.3 V). The wireless module, when connected using SPI, can be detected by reading a register with a known value and comparing those.

9.7.1 USB Connection

GEX uses vid:pid `1209:4c60` and the wireless gateway `1209:4c61`. The USB interface uses the CDC/ACM USB class (4.3.2) and consists of two bulk endpoints with a payload size of up to 64 bytes.

9.7.2 Communication UART

The parameters of the communication UART (such as the baud rate) are defined in `SYSTEM.INI`. It's mapped to pins PA2 and PA3; this is useful with STM32 Nucleo boards that don't include a User USB connector, but provide a USB-serial bridge using the on-board ST-Link programmer, connected to those pins.

This is identical to the USB connection from the PC application's side, except a physical UART is necessarily slower and does not natively support flow control. The use of the Xon and Xoff software flow control is not practical with binary messages that could include those bytes by accident, and the ST-Link USB-serial adapter does not implement hadware flow control.

9.7.3 Wireless Connection

The wireless connection uses an on-board communication module and a separate device, a wireless gateway, that connects to the PC. The wireless gateway is interfaced differently from the GEX board itself, but it also shows as a virtual serial port on the host PC. This is required to allow communicating with the gateway itself through the CDC/ACM interface in addition to addressing the end devices.

This interface will be explained in more detail in chapter 11.

9.8 Message Passing

One of the key functions of the core framework is to deliver messages from the host PC to the right units. This functionality resides above the framing protocol, which will be described in chapter 10.

A message that is not a response in a multi-part session (this is handled by the framing library) is identified by its Type field. Two main groups of messages exist: *system messages* and *unit messages*. System messages can access the INI files, query a list of the available units, restart the module etc. Unit messages are addressed to a particular unit by their callsign (see 9.5), and their payload format is defined by the unit driver. The framework reads the message type, then the callsign byte, and tries to find a matching unit in the unit list. If no unit with the callsign is found, an error response is sent back, otherwise the unit driver is given the message to handle it as required.

The framework provides one more messaging service to the units: event reporting. An asynchronous event, such as an external interrupt, an ADC trigger or an UART data reception needs to be reported to the host. This message is annotated by the unit callsign so the user application knows it's origin.

9.9 Interrupt Routing

Interrupts are an important part of almost any embedded application. They provide a way to rapidly react to asynchronous external or internal events, temporarily leaving the

main program, jumping to an interrupt handler routine, and then returning back after the event is handled. Interrupts are also the way FreeRTOS implements multitasking without a multi-core processor.

In the Cortex-M0-based STM32F072, used in the initial GEX prototypes, the interrupt handlers table, defining which routine is called for which interrupt, is stored in the program memory and can't be changed at run-time. This is a complication for the modular structure of GEX where different unit drivers may use the same peripheral, and we would want to dynamically assign the interrupt handlers based on the active configuration. Let's have a look at an interrupt handler, in this case handling four different DMA channels, as is common in STM32 microcontrollers:

```
void DMA1_Channel4_5_6_7_IRQHandler(void)
{
    if (LL_DMA_IsActiveFlag_GI4(DMA1)) { /* handle DMA1 channel 4 */ }
    if (LL_DMA_IsActiveFlag_GI5(DMA1)) { /* handle DMA1 channel 5 */ }
    if (LL_DMA_IsActiveFlag_GI6(DMA1)) { /* handle DMA1 channel 6 */ }
    if (LL_DMA_IsActiveFlag_GI7(DMA1)) { /* handle DMA1 channel 7 */ }
}
```

It is evident that multiple units might need to use the same interrupt handler, even at the same time, since each DMA channel is configured, and works, independently. GEX implements a redirection scheme to accomplish such interrupt sharing: All interrupt handlers are defined in one place, accompanied by a table of function pointers. When a unit driver wants to register an interrupt handler, it stores a pointer to it in this redirection table. Then, once an interrupt is invoked, the common handler checks the corresponding entry in the table and calls the referenced routine, if any. Conversely, when a unit driver deinitializes a unit, it removes all interrupt handlers it used, freeing the redirection table slots for other use.

Chapter 10

Communication Protocol

GEX can be controlled through a hardware UART, the USB or over a wireless link. To minimize the firmware complexity, all the three connection methods are handled by the same protocol stack and are interchangeable.

The communication is organized in transactions. A transaction consists of one or more messages going in either direction. Messages can be stand-alone, or chained with a response or a follow-up message using the transaction ID. Both peers, GEX and the client application running on the PC, are equal in the communication: either side can independently initiate a transaction at any time.

GEX uses a framing library *TinyFrame*, developed likewise by the author but kept as a separate project for easier re-use in different applications. The library implements frame building and parsing, checksum calculation and a system of message listeners.

10.1 Frame Structure

Message frames have the following structure:

	Header					Body	
Field	**SOF**	**Frame ID**	**Payload Length**	**Frame type**	**Header checksum**	**Payload**	**Payload checksum**
Bytes	1	2	2	1	1	...	1

The field widths shown here are those used in GEX; TinyFrame is flexible and the data type of all fields can be customized, as well as the checksum type. The SOF byte is always 0x01.

Frame ID, which could be better described as *Transaction ID*, uniquely identifies each transaction. The most significant bit is set to a different value in each peer to avoid ID conflicts, and the rest of the ID field is incremented with each initiated transaction.

10.2 Message Listeners

After sending a message that should receive a response, the peer registers an *ID listener* with the ID of the sent message. A response reuses the original frame ID and when it is received, this listener is called to process it. ID listeners can also be used to receive multi-part messages re-using the original ID.

Frame type describes the payload and does not have any prescribed format; the values are defined by application (here, GEX). A *type listener* may be registered to handle all incoming messages with a given frame type. It works in a similar way to an ID listener and has a lower priority.

Each message can be handled by only one listener, unless it explicitly requests the message to be passed on to a lower priority one. Messages unhandled by any listener are given to a default listener, which can e.g. write an error to a debug log.

10.3 Designated Frame Types in GEX

The following table lists all frame types used by GEX. It is divided into four logical sections: General, Bulk Read/Write, Unit Access, and Settings.

Frame type	**Function**	**Note**
0x00	Success	*Payload depends on context*
0x01	Ping	*GEX responds with Success and its version string*
0x02	Error	*Payload contains the error message*
0x03	Bulk Read Offer	*An offer of data to read using* 0x04
0x04	Bulk Read Poll	*Requesting to read a block of data*
0x05	Bulk Write Offer	*An offer to receive a bulk write transaction*
0x06	Bulk Data	*Used for both reading and writing*
0x07	Bulk End	*Marks the last "Bulk Data" frame*
0x08	Bulk Abort	
0x10	Unit Request	*Request from PC to a unit*
0x11	Unit Report	*Spontaneous event generated by a unit*
0x20	List Units	*Read a list of all instantiated units*
0x21	INI Read	*Request a bulk read transaction of an INI file*
0x22	INI Write	*Request a bulk write transaction of an INI file*
0x23	Persist Config	*Write updated configuration to Flash*

10.4 Bulk Read and Write Transactions

The bulk read and write transactions are generic, multi-message exchanges which are used to transfer the INI configuration files. They could additionally be used by some future unit requiring to transfer a large amount of data (e.g. to read image data from a camera).

The reason for splitting a long file into multiple messages, rather than sending it all in one, lies in the hardware limitations of the platform, specifically its small amount of RAM (the STM32F072 has only 16 kB). A message cannot be processed until its payload checksum is received and verified; however, the configuration file can have several kilobytes, owning to the numerous explanatory comments, which would require a prohibitively large data buffer. The chunked transaction could, additionally, be extended to support message re-transmission on timeout without sending the entire file again.

A read or write transaction can be aborted by a frame 0x08 (Bulk Abort) at any time, though aborting a write transaction may leave the configuration in a corrupted state. As hinted in the introduction of this chapter, a transaction is defined by sharing a common frame ID. Thus, all frames in a bulk transaction must have the same ID, otherwise the ID listeners won't be called for the subsequent messages, and the transaction will time out.

Figure 10.1 shows a diagram of the bulk read and write data flow.

10.4.1 Bulk Read

To read an INI file, we first send a frame 0x21 (INI Read), specifying the target file in the payload:

```
struct Payload_INI_Read {
    uint8_t filenum; // 0 - UNITS.INI, 1 - SYSTEM.INI
};
```

What follows is a standard bulk read transaction with the requested file. GEX offers the file for reading with a frame 0x03 (Bulk Read Offer):

```
struct Payload_BulkReadOffer {
    uint32_t total_length;   // full size of the file in bytes
    uint32_t max_chunk_size; // largest chunk that can be read at once
};
```

Now we can proceed to read the file using 0x04 (Bulk Read Poll), which is always responded to with 0x06 (Bulk Data), or 0x07 (Bulk End) if this was the last frame. Data frames have only the useful data as their payload.

The 0x04 (Bulk Read Poll) payload specifies how many bytes we want to read:

```
struct Payload_BulkReadPoll {
    uint32_t max_chunk_size; // how many bytes to read
};
```

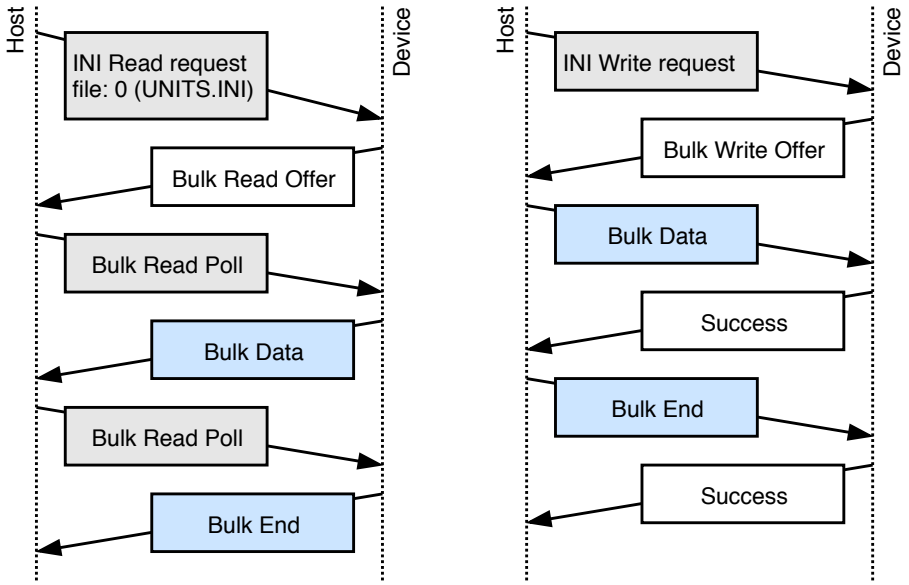


Figure 10.1: A diagram of the bulk read and write transaction.

10.4.2 Bulk Write

To overwrite an INI file, we first send a frame 0x22 (INI Write), specifying its size in the payload. Which file is written is detected automatically from the first INI section.

```
struct Payload_INI_Write {
    uint32_t total_length; //  file size in bytes
};
```

The write request is confirmed by a frame 0x05 (Bulk Write Offer):

```
struct Payload_BulkWriteOffer {
    uint32_t total_length;   // the expected file size in bytes
    uint32_t max_chunk_size; // largest chunk that can be written at once
};
```

We can now send the file as a series of frames 0x06 (Bulk Data), or 0x07 (Bulk End) in the last frame. Each written chunk is confirmed by 0x00 (Success).

10.4.3 Persisting the Changed Configuration to Flash

The written INI file is immediately parsed and the settings are applied. However, those changes are not persistent: they exist only in RAM and will be lost when the module restarts. To save the current state to Flash, issue a frame 0x23 (Persist Config). This has the same effect as pressing the LOCK button (or replacing the LOCK jumper) when the INI files are edited using the virtual mass storage.

It should be noted that after flashing a firmware, the Flash control registers may remain in an unexpected state and the module must first be manually restarted before attempting to persist settings. Otherwise an assertion will fail and the module is restarted by a watchdog, losing the temporary changes.

10.5 Reading a List of Units

The frame 0x20 (List Units) requests a list of all available units in the GEX module. The list includes all units' callsigns, names and types. The response payload has the following format (in pseudocode, as it can't be expressed as a C struct like the previous examples):

```
struct {
    uint8_t count;
    for all units {
        uint8_t callsign;
        cstring unit_name;   // 0-terminated char array
        cstring driver_name;
    }
}
```

10.6 Unit Requests and Reports

Frame types 0x10 (Unit Request) and 0x11 (Unit Report) are dedicated to messages sent to and by unit instances. Each has a fixed header (*inside the payload*) followed by unit-specific data.

10.6.1 Unit Requests

Unit requests deliver a message from the host to a unit instance. Unit drivers implements different commands, each with its own payload structure. The frame 0x10 (Unit Request) has the following structure:

```
struct Payload_UnitRequest {
    uint8_t callsign;
    uint8_t command;   // handled by the unit driver
    uint8_t payload[]; // size and content depend on the command
};
```

The most significant bit of the command byte (0x80) has a special meaning: when set, the message delivering routine responds with 0x00 (Success) after the command completes, unless an error occurred. That is used to get a confirmation that the message was delivered and the module operates correctly (as opposed to e.g. a lock-up resulting in a watchdog reset). Requests which normally generate a response (e.g. reading a value from the unit) should not be sent with this bit set. As a result of this special treatment of the highest bit, there can be only 127 different commands per unit.

10.6.2 Unit Reports

Several unit types can produce asynchronous events, such as reporting a pin change or a triggering condition. The event is timestamped and sent with a frame type 0x11 (Unit Report):

```
struct Payload_UnitRequest {
    uint8_t callsign;
    uint8_t report_type; // defines the payload structure
    uint64_t timestamp;  // microseconds since power-on
    uint8_t payload[];   // size and content depend on the report type
};
```

Chapter 11

Wireless Interface

Four methods of a wireless connection have been considered: Bluetooth (e.g. with CC2541), WiFi with ESP8266, LoRA or GFSK-modulated 868 MHz link with SX1276, and a 2.4 GHz link with nRF24L01+. Bluetooth was dismissed early for its complexity and ESP8266 for its high consumption in continuous reception mode, although both solutions might be viable for certain applications and with more time for evaluation.

The Semtech SX1276 and Nordic Semiconductor nRF24L01+ transceivers have both been tested using the first GEX prototype, confirming its usefulness as a hardware development tool, and it's been confirmed they could fulfill the requirements of the application.

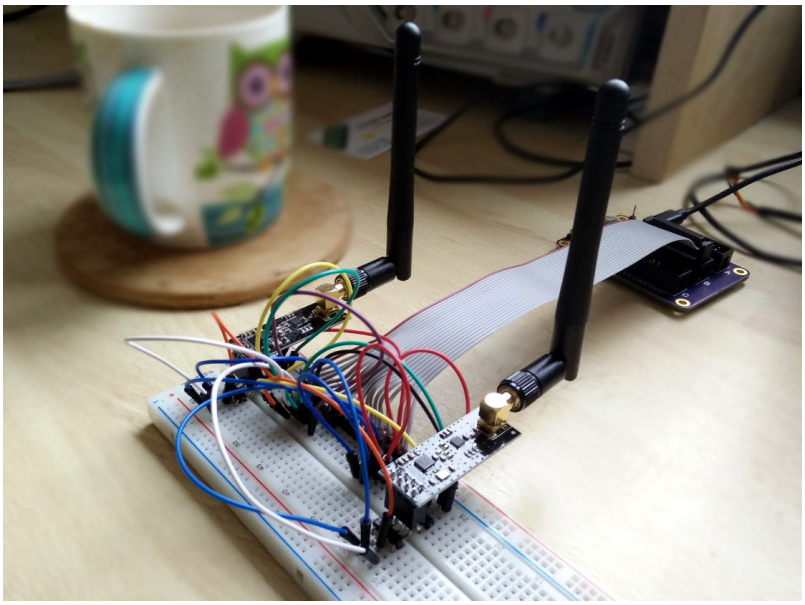


Figure 11.1: Test setup with a GEX prototype controlling two nRF24L01+ modules

11.1 Comparing SX1276 vs. nRF24L01+

The two transceivers are compared in table 11.1. It's apparent that each has its strengths and weaknesses.

SX1276 supports additional modulation modes, including a proprietary LoRa scheme with a frequency-hopping spread spectrum modulation that can be received at a distance up

Parameter	SX1276	nRF24L01+
Connection	SPI (4 pins) + up to 6 IRQ	SPI (4 pins), CE, IRQ
Frequency band	868 MHz or 433 MHz	2.4 GHz
Data rate	up to 300 kbps	250–2000 kbps
Modulation	(G)FSK, (G)MSK, OOK, LoRa	GFSK
Range (est.)	2 km to 20 km LOS	up to 1 km LOS
Consumption Rx	10.8–12 mA	12.6–13.5 mA
Consumption Tx	20–120 mA	7–11.3 mA
Idle power (max)	1 μA sleep, 2 mA stand-by	0.9 μA sleep, 320 μA stand-by
Max packet size	300 bytes	32 bytes
Reset	NRESET pin	Vdd disconnect
Extra	LoRa FHSS, packet engine	ShockBurst protocol engine
Price	$7.3	$1.6

Table 11.1: Comparison of the SX1276 and nRF24L01+ wireless transceivers (price from DigiKey @ 10 pieces, May 6th 2018)

to 20 km in ideal conditions. The long-range capability is reflected in a higher consumption during transmission. However, its consumption in receiver mode is slightly lower than that of the nRF24L01+.

nRF24L01+ provides higher data rates at short distances. Its power consumption is comparable or lower than that of the SX1276. It lacks a dedicated reset pin, but that can be easily worked around using an external transistor to momentarily disconnect its Vdd pin.

Both devices implement some form of a packet engine with error checking; that of the nRF24L01+, called ShockBurst, is more advanced as it implements acknowledgment responses and automatic re-transmission, leading to a potentially more robust communication without an additional overhead on the side of the microcontroller.

11.2 Integration of the nRF24L01+ into GEX

The nRF24L01+ was selected to be integrated into GEX thanks to its inclusion of the ShockBurst engine, higher possible data rates and significantly lower price. The SX1276, nonetheless, remains an interesting option that could be used as an alternative in the future, should the need for a long range communication arise.

A separate device, the *GEX wireless gateway*, was developed to provide the PC connection to a nRF24L01+ module. It is based on the STM32F103 microcontroller in its smallest package (LQFP48), selected for it's low cost and good availability.

11.2.1 The Wireless Gateway Protocol

The gateway presents itself to the host as a CDC/ACM device, much like the GEX modules themselves (here called *nodes*) when connected over USB. It implements a simple protocol which encapsulates the binary data sent to or from a connected node. The wrapped GEX protocol (chapter 10) remains unchanged.

The gateway has a 4-byte network ID, a number derived from the microcontroller's unique ID. The network ID must be entered into all nodes that wish to communicate with it. Additionally, each module must be assigned a unique 1-byte number, which, together with the four network ID bytes, uniquely identifies the node. The gateway can connect to up to 6 nodes at once.

All messages sent to or from the gateway are a multiple of 64 bytes long, padded with zeros if shorter. The message starts with a control byte, determining the message type (table 11.2).

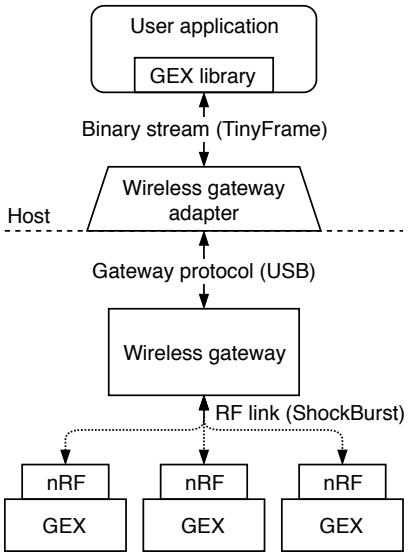


Figure 11.2: A block diagram of the wireless connection

Function	Message structure
Send a message	'm' (`u8`)address (`u16`)length (`u8[]`)payload
Restart & remove nodes	'r'
Add nodes	'n' (`u8`)count (`u8[]`)addresses
Get the network ID	'i'
Response to 'i'	0x01 (`u8[4]`)net_id
Received message	0x02 (`u8`)address (`u8`)length (`u8[]`)data

Table 11.2: Wireless gateway commands and messages; control characters in the printable range are shown as ASCII

The gateway may be restarted using the 'r' command when the PC application starts. Then it adds all node addresses with the 'n' command and can begin communication. The 'i' command, reading the network ID, can additionally be used as a ping command to verify the USB connection.

Add additional commands. Also remove magic bytes form the real messages, they're useless

Chapter 12

Units Overview and API

This chapter describes all functional blocks (units) implemented in GEX at the time of publication of this work. Each unit supports a different set of binary commands and events. The term "unit" will be used to refer to both unit types (drivers) or their instances, where the distinction is not important.

Each unit's description will be accompanied by a corresponding snippet from the configuration file, and a list of supported commands and events. When the command's request payload is empty, it's omitted from the table. The same applies to commands with no response, in which case adding 0x80 to the command number triggers a SUCCESS response if the command succeeds.

12.1 Naming Conventions and Common Principles

12.1.1 Unit Naming

Unit types are named in uppercase (e.g. SPI, 1WIRE, NPX) in the INI file and in the list of units. Unit instances can be named in any way the user desires; Using lowercase makes it easier to distinguish them from unit types. It is advisable to use descriptive names, e.g. not "pin1" but rather "button".

12.1.2 Packed Pin Access

Several units facilitate an access to a group of GPIO pins, such as the digital input and output units, or the SPI unit's slave select pins. The STM32 microcontroller's ports have 16 pins each, most of which can be configured to one of several alternate functions (e.g. SPI, PWM outputs, ADC input). As a consequence, it's common to be left with a discontiguous group of pins after assigning all the alternate functions needed by an application.

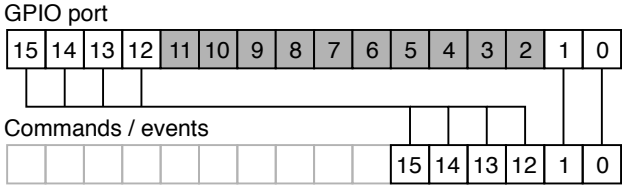


Figure 12.1: Pin packing

For instance, we could only have the pins 0, 1, 12–15 available on a GPIO port. GEX provides a helpful abstraction to bridge the gaps in the port: The selected pins are packed together and represented, in commands and events, as a block of six pins (0x3F) instead of their original positions in the register (0xF003). This scheme is shown in figure 12.1. The translation is done in the unit driver and works transparently, as if the block of pins had no gaps—all the referenced pins are updated simultaneously without glitches. Where pin numbers are used, the order in the packed word should be provided—in our example, that would be 0–5, counting from the least significant bit.

12.2 Digital Output

The digital output unit provides a write access to one or more pins of a GPIO port. This unit additionally supports pulse generation on any of its pins. This is implemented in software with the timing derived from the system timebase, as the hardware timer outputs, otherwise used for PWM or pulse generation, are available only on several dedicated pins. The timing code is optimized to reduce jitter.

Measure jitter and add it here

12.2.1 Digital Output Configuration

```
[DO:out@1]
# Port name
port=A
# Pins (comma separated, supports ranges)
pins=0
# Initially high pins
initial=
# Open-drain pins
open-drain=
```

12.2.2 Digital Output Commands

Code	Function	Structure
0	**WRITE** Write to all pins	*Request:* • `u16` new value
1	**SET** Set selected pins to 1	*Request:* • `u16` pins to set
2	**CLEAR** Set selected pins to 0	*Request:* • `u16` pins to clear
3	**TOGGLE** Toggle selected pins	*Request:* • `u16` pins to toggle

Code	Function	Structure
4	**PULSE** Generate a pulse on the selected pins. The μs scale may be used only for 0–999 μs.	*Request:* • `u16` pins to pulse • `u8` active level (0, 1) • `u8` scale: 0-ms, 1-μs • `u16` duration

12.3 Digital Input

The digital input unit is the input counterpart of the digital output unit.

In addition to reading the immediate digital levels of the selected pins, this unit can generate asynchronous events on a pin change. The state of the entire input port, together with a microsecond timestamp (as is the case for all asynchronous events), is reported to the host either on a rising, falling, or any pin change.

The pin change event can be configured independently for each pin. In order to receive a pin change event, it must be armed first; The pin can be armed for a single event, or it may be re-armed automatically with a hold-off time. It's further possible to automatically arm selected pin triggers on start-up.

12.3.1 Digital Input Configuration

```
[DI:in@2]
# Port name
port=A
# Pins (comma separated, supports ranges)
pins=0
# Pins with pull-up
pull-up=
# Pins with pull-down
pull-down=

# Trigger pins activated by rising/falling edge
trig-rise=
trig-fall=
# Trigger pins auto-armed by default
auto-trigger=
# Triggers hold-off time (ms)
hold-off=100
```

12.3.2 Digital Input Events

Code	Function	Structure
0	**PIN_CHANGE** A pin change event. The payload includes a snapshot of all configured pins captured immediately after the change was registered.	*Payload:* • `u16` changed pins • `u16` port snapshot

12.3.3 Digital Input Commands

Code	Function	Structure
0	**READ** Read the pins	*Response:* • `u16` pin states
1	**ARM_SINGLE** Arm for a single event	*Request:* • `u16` pins to arm
2	**ARM_AUTO** Arm with automatic re-arming after each event	*Request:* • `u16` pins to arm
3	**DISARM** Dis-arm selected pins	*Request:* • `u16` pins to dis-arm

12.4 SIPO (Shift Register) Unit

The shift registers driver unit is designed for the loading of data into *serial-in, parallel-out* (SIPO) shift registers, such as 74HC4094 or 74HC595. Those are commonly used to control segmented LED displays, LED matrices etc.

This unit handles both the *Shift* and *Store* signals and is capable of loading multiple shift registers simultaneously, reducing visible glitches in the display. It's also possible to set the data lines to arbitrary level(s) before sending the Store pulse, which can be latched and used for some additional feature of the LED display, such as brightness control.

12.4.1 SIPO Configuration

```
[SIPO:display@9]
# Shift pin & its active edge (1-rising,0-falling)
shift-pin=A1
shift-pol=1
# Store pin & its active edge
store-pin=A0
store-pol=1
```

```
# Clear pin & its active level
clear-pin=A2
clear-pol=0
# Data port and pins
data-port=A
data-pins=3
```

12.4.2 SIPO Commands

Code	Function	Structure
0	**WRITE** Load the shift registers and leave the data outputs in the "trailing data" state before sending the Store pulse.	*Request:* • `u16` trailing data • For each output (same size) – `u8[]` data to load
1	**DIRECT_DATA** Directly write to the data pins (same like the DO unit's WRITE command)	*Request:* • `u16` values to write
2	**DIRECT_CLEAR** Pulse the Clear pin, erasing the registers' data	
3	**DIRECT_SHIFT** Pulse the Shift pin	
4	**DIRECT_STORE** Pulse the Store pin	

12.5 NeoPixel Unit

The NeoPixel unit implements the protocol needed to control a digital LED strip with WS2812, WS2811, or compatible LED driver chips. The protocol timing is implemented in software, therefore it is available on any GPIO pin of the module.

The color data can be loaded in five different format: as packed bytes, or as the little-endian or big-endian encoding of colors in the 32-bit format 0x00RRGGBB or 0x00BBGGRR. This data format is convenient when the colors are already represented by an array of 32-bit integers.

12.5.1 NeoPixel Configuration

```
[NPX:neo@3]
# Data pin
pin=A0
```

```
# Number of pixels
pixels=32
```

12.5.2 NeoPixel Commands

Code	Function	Structure
0	**CLEAR** Switch all LEDs off (sets them to black)	
1	**LOAD** Load a sequence of R,G,B bytes	*Request:* • For each LED: – `u8` red – `u8` green – `u8` blue
4	**LOAD_U32_ZRGB** Load 32-bit big-endian 0xRRGGBB (0,R,G,B)	*Request:* • `u32[]` color data BE
5	**LOAD_U32_ZBGR** Load 32-bit big-endian 0xBBGGRR (0,B,G,R)	*Request:* • `u32[]` color data BE
6	**LOAD_U32_RGBZ** Load 32-bit little-endian 0xBBGGRR (R,G,B,0)	*Request:* • `u32[]` color data LE
7	**LOAD_U32_BGRZ** Load 32-bit little-endian 0xRRGGBB (B,G,R,0)	*Request:* • `u32[]` color data LE
10	**GET_LEN** Get number of LEDs in the strip	*Response:* • `u16` number of LEDs

12.6 SPI Unit

The SPI unit provides access to one of the microcontroller's SPI peripherals. It can be configured to use any of the different speeds, clock polarity and phase settings available in its control registers.

The unit handles up to 16 slave select (NSS) signals and supports message multi-cast (addressing more than one slaves at once). Protection resistors should be used if a multi-cast transaction is issued with MISO connected.

The QUERY command of this unit, illustrated by figure 12.2, is flexible enough to support all types of SPI transactions: read-only, write-only, and read-write with different request and response lengths. The slave select pin is held low during the entire transaction.

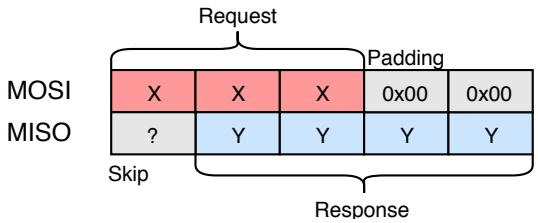


Figure 12.2: SPI transaction using the QUERY command

12.6.1 SPI Configuration

```
[SPI:spi@5]
# Peripheral number (SPIx)
device=1
# Pin mappings (SCK,MISO,MOSI)
#  SPI1: (0) A5,A6,A7     (1) B3,B4,B5
#  SPI2: (0) B13,B14,B15
remap=0
# Prescaller: 2,4,8,...,256
prescaller=64
# Clock polarity: 0,1 (clock idle level)
cpol=0
# Clock phase: 0,1 (active edge, 0-first, 1-second)
cpha=0
# Transmit only, disable MISO
tx-only=N
# Bit order (LSB or MSB first)
first-bit=MSB
# SS port name
port=A
# SS pins (comma separated, supports ranges)
pins=0
```

12.6.2 SPI Commands

Code	Function	Structure
0	**QUERY** Exchange bytes with a slave device	*Request:* • `u8` slave number 0–16 • `u16` response padding • `u16` response length • `u8[]` bytes to write *Response:* • `u8[]` received bytes

Code	Function	Structure
1	**MULTICAST** Send a message to multiple slaves at once. The address is a bit map (e.g. 0x8002 = slaves 1 and 15).	*Request:* • `u16` addressed slaves • `u8[]` bytes to write

12.7 I2C Unit

The I2C unit provides access to one of the microcontroller's I2C peripherals. It can be configured to use either of the three speeds (Standard, Fast and Fast+) and supports both 10-bit and 7-bit addressing. 10-bit addresses can be used in commands by setting their highest bit (0x8000), as a flag to the unit.

12.7.1 I2C Configuration

```
[I2C:d@4]
# Peripheral number (I2Cx)
device=1
# Pin mappings (SCL,SDA)
#  I2C1: (0) B6,B7    (1) B8,B9
#  I2C2: (0) B10,B11  (1) B13,B14
remap=0

# Speed: 1-Standard, 2-Fast, 3-Fast+
speed=1
# Analog noise filter enable (Y,N)
analog-filter=Y
# Digital noise filter bandwidth (0-15)
digital-filter=0
```

12.7.2 I2C Commands

Code	Function	Structure
0	**WRITE** Raw write transaction	*Request:* • `u16` slave address • `u8[]` bytes to write

Code	Function	Structure
1	**READ** Raw read transaction	*Request:* • `u16` slave address • `u16` number of read bytes *Response:* • `u8[]` received bytes
2	**WRITE_REG** Write to a slave register. Sends the register number and the data in the same I2C transaction. Multiple registers can be written to slaves supporting auto-increment.	*Request:* • `u16` slave address • `u8` register number • `u8[]` bytes to write
3	**READ_REG** Read from a slave register. Writes the register number and issues a read transaction of the given length. Multiple registers can be read from slaves supporting auto-increment.	*Request:* • `u16` slave address • `u8` register number • `u16` number of read bytes *Response:* • `u8[]` received bytes

12.8 USART Unit

The USART unit provides access to one of the microcontroller's USART peripherals. All USART parameters can be configured to match the application's needs. The peripheral is capable of driving RS485 transceivers with the Driver Enable (DE) output for switching between reception and transmission.

The unit implements asynchronous reception and transmission using DMA and a circular buffer. Received data is sent to the host in asynchronous events when either half of the buffer is filled, or after a fixed timeout from the last received byte.

12.8.1 USART Configuration

```
[USART:ser@6]
# Peripheral number (UARTx 1-4)
device=1
# Pin mappings (TX,RX,CK,CTS,RTS/DE)
#  USART1: (0) A9,A10,A8,A11,A12   (1) B6,B7,A8,A11,A12
#  USART2: (0) A2,A3,A4,A0,A1      (1) A14,A15,A4,A0,A1
#  USART3: (0) B10,B11,B12,B13,B14
#  USART4: (0) A0,A1,C12,B7,A15    (1) C10,C11,C12,B7,A15
remap=0

# Baud rate in bps (eg. 9600)
```

```
baud-rate=115200
# Parity type (NONE, ODD, EVEN)
parity=NONE
# Number of stop bits (0.5, 1, 1.5, 2)
stop-bits=1
# Bit order (LSB or MSB first)
first-bit=LSB
# Word width (7,8,9) - including parity bit if used
word-width=8
# Enabled lines (RX,TX,RXTX)
direction=RXTX
# Hardware flow control (NONE, RTS, CTS, FULL)
hw-flow-control=NONE

# Generate serial clock (Y,N)
clock-output=N
# Clock polarity: 0,1
cpol=0
# Clock phase: 0,1
cpha=0

# Generate RS485 Driver Enable signal (Y,N) - uses RTS pin
de-output=N
# DE active level: 0,1
de-polarity=1
# DE assert time (0-31)
de-assert-time=8
# DE clear time (0-31)
de-clear-time=8
```

12.8.2 USART Events

Code	Function	Structure
0	**DATA_RECEIVED** Data was received on the serial port.	*Payload:* • `u8[]` received bytes

12.8.3 USART Commands

Code	Function	Structure
0	**WRITE** Add data to the transmit buffer. Sending is asynchronous, but the command may wait for free space in the DMA buffer.	*Request:* • `u8[]` bytes to write
1	**WRITE_SYNC** Add data to the transmit buffer and wait for the transmission to complete.	*Request:* • `u8[]` bytes to write

12.9 1-Wire Unit

The 1-Wire unit implements the Dallas Semiconductor's 1-Wire protocol, most commonly used to interface smart thermometers (DS18x20). The protocol is explained in section 7.4.

This unit implements the ROM Search algorithm that is used to find the unique IDs of all 1-Wire devices connected to the bus. The algorithm can find up to 32 devices in one run; More devices can be found by issuing the SEARCH_CONTINUE command.

Devices are addressed using their 64-bit (8-byte) identifiers. When only one device is connected, the value 0 may be used and the addressing will be skipped. Its ID may be recovered using the READ_ADDR command or by the search algorithm.

12.9.1 1-Wire Configuration

```
[1WIRE:ow@7]
# Data pin
pin=A0
# Parasitic (bus-powered) mode
parasitic=N
```

12.9.2 1-Wire Commands

Code	Function	Structure
0	**CHECK_PRESENCE** Test if there are any devices attached to the bus.	*Response:* • `u8` presence detected (0, 1)
1	**SEARCH_ADDR** Start the search algorithm.	*Response:* • `u8` should continue (0, 1) • `u64[]` device addresses

Code	Function	Structure
2	**SEARCH_ALARM** Start the search algorithm, finding only devices in an alarm state.	*Response:* • `u8` should continue (0, 1) • `u64[]` device addresses
3	**SEARCH_CONTINUE** Continue a previously started search	*Response:* • `u8` should continue (0, 1) • `u64[]` device addresses
4	**READ_ADDR** Read a device address (single device only)	*Response:* • `u64` device address
10	**WRITE** Write bytes to a device.	*Request:* • `u64` device address • `u8[]` bytes to write
11	**READ** Write a request and read response.	*Request:* • `u64` device address • `u16` read length • `u8` verify checksum (0, 1) • `u8[]` request bytes *Response:* • `u8[]` read bytes
20	**POLL_FOR_1** Wait for a READY status, used by DS18x20. Not available in parasitic mode. Responds with SUCCESS after all devices are ready.	

12.10 Frequency Capture Unit

The frequency capture unit implements both the frequency measurement methods explained in section 8.1: direct and reciprocal.

The unit has several operational modes: idle, reciprocal continuous, reciprocal burst, direct continuous, direct burst, free counting, and single pulse. Burst mode is an on-demand measurement with possible averaging. Continuous mode doesn't support averaging, but the latest measurement can be read at any time without a delay.

12.10.1 Value Conversion Formulas

Several of the features implemented in this unit would require floating point arithmetic to provide the measured value in the desired units (Hz, seconds). That is not available in Cortex-M0, only as a software implementation. The calculation is left to the client in order

to save Flash space that would be otherwise used by the the arithmetic functions. This arrangement also avoids rounding errors and a possible loss of precision.

Reciprocal (Indirect) Measurement

Period (in seconds) is computed as:

$$T = \frac{\text{period\_sum}}{f_{\text{core,MHz}} \cdot 10^6 \cdot \text{n\_periods}}$$

The frequency is obtained by simply inverting it:

$$f = T^{-1}$$

The average duty cycle is computed as the ratio of the sum of active-level pulses and the sum of all periods:

$$\text{average\_duty} = \frac{\text{ontime\_sum}}{\text{period\_sum}}$$

Direct Measurement

The frequency can be derived from the pulse count and measurement time using its definition (t_{ms} is measurement time in milliseconds):

$$f = \frac{1000 \cdot \text{count} \cdot \text{prescaller}}{t_{\text{ms}}}$$

12.10.2 Frequency Capture Configuration

```
[FCAP:j@10]
# Signal input pin - one of:
#  Full support:  A0, A5, A15
#  Indirect only: A1, B3
pin=A0

# Active level or edge (0-low,falling; 1-high,rising)
active-level=1
# Input filtering (0-15)
input-filter=0
# Pulse counter pre-divider (1,2,4,8)
direct-presc=1
# Pulse counting interval (ms)
direct-time=1000
```

```
# Mode on startup: N-none, I-indirect, D-direct, F-free count
initial-mode=N
```

12.10.3 Frequency Capture Commands

Some commands include optional parameter setting. Using 0 in the field keeps the previous value. Those fields are marked with *.

Code	Function	Structure
0	**STOP** Stop all measurements, go idle	
1	**INDIRECT_CONT_START** Start a repeated reciprocal measurement	
2	**INDIRECT_BUTST_START** Start a burst of reciprocal measurements	*Request:* • `u16` number of periods *Response:* • `u16` core speed (MHz) • `u16` number of periods • `u64` sum of all periods (ticks) • `u16` sum of on-times (ticks)
3	**DIRECT_CONT_START** Start a repeated direct measurement	*Request:* • `u16` *measurement time • `u8` *prescaller (1, 2, 4, 8)
4	**DIRECT_BURST_START** Start a single direct measurement. Longer capture time may help increase accuracy for stable signals.	*Request:* • `u16` *measurement time (ms) • `u8` *prescaller (1, 2, 4, 8) *Response:* • `u8` prescaller • `u16` measurement time (ms) • `u32` pulse count
5	**FREECOUNT_START** Clear and start the pulse counter	*Request:* • `u8` *prescaller (1,2,4,8)
6	**MEASURE_SINGLE_PULSE** Measure a single pulse of the active level. Waits for a rising edge.	*Response:* • `u16` core speed (MHz) • `u32` pulse length (ticks)
7	**FREECOUNT_CLEAR** Read and clear the pulse counter.	*Response:* • `u32` previous counter value

Code	Function	Structure
10	**INDIRECT_CONT_READ** Read the latest value from the continuous reciprocal measurement, if running.	*Response:* • `u16` core speed (MHz) • `u32` period length (ticks) • `u32` on-time (ticks)
11	**DIRECT_CONT_READ** Read the latest value from the continuous direct measurement, if running.	*Response:* • `u8` prescaller • `u16` measurement time (ms) • `u32` pulse count
12	**FREECOUNT_READ** Read the pulse counter value	*Response:* • `u32` pulse count
20	**SET_POLARITY** Set pulse polarity (active level)	*Response:* • `u8` polarity (0,1)
21	**SET_PRESCALLER** Set prescaller for the direct mode	*Response:* • `u8` prescaller (1,2,4,8)
22	**SET_INPUT_FILTER** Set input filtering (a hardware feature designed to ignore glitches)	*Response:* • `u8` filtering factor (0-15, 0=off)
23	**SET_DIR_MSEC** Set direct measurement time	*Response:* • `u16` measurement time (ms)
30	**RESTORE_DEFAULTS** Restore all run-time adjustable parameters to their configured default values	

12.11 ADC Unit

The analog/digital converter unit is one of the most complicated units implemented in the project. The unit can measure the voltage on an input pin, either as its immediate value, or averaged with exponential forgetting. Isochronous sampling is available as well: It's possible to capture a fixed-length block of data on demand, or as a response to a triggering condition on any of the enabled input pins. The ADC must continuously sample the inputs to make the averaging and level based triggering possible; As a consequence, a pre-trigger buffer is available that can be read together with the block of samples following a trigger. The ADC unit can also be switched to a continuous streaming mode.

It's possible to activate any number of the 16 analog inputs of the ADC peripheral simultaneously. The maximum continuous sampling frequency, which reaches 70 ksps with one channel, lowers with an increasing number of enabled channels as the amount of data to transfer to the host increases.

12.11.1 ADC Configuration

```
[ADC:adc@8]
# Enabled channels, comma separated
#  0  1  2  3  4  5  6  7    8  9   10 11 12 13 14 15   16    17
# A0 A1 A2 A3 A4 A5 A6 A7   B0 B1   C0 C1 C2 C3 C4 C5   Tsens Vref
channels=16

# Sampling time (0-7)
sample_time=2
# Sampling frequency (Hz)
frequency=1000

# Sample buffer size
# - shared by all enabled channels
# - defines the maximum pre-trigger size (divide by # of channels)
# - captured data is sent in half-buffer chunks
# - buffer overrun aborts the data capture
buffer_size=256

# Enable continuous sampling with averaging
# Caution: This can cause DAC output glitches
averaging=Y
# Exponential averaging coefficient (permil, range 0-1000 ~ 0.000-1.000)
# - used formula: y[t]=(1-k)*y[t-1]+k*u[t]
# - not available when a capture is running
avg_factor=500
```

12.11.2 ADC Events

Code	Function	Structure
50	**TRIGGERED** The first event generated when a triggering condition occurs. The payload includes pre-trigger and the transaction continues with a sequence of CAPTURE events sharing the same frame ID. The serial number is incremented with each stream chunk and can be used to detect lost data frames.	*Payload:* • `u32` pre-trigger length • `u8` triggering edge (1-falling, 2-rising, 3-forced) • `u8` stream serial number • `u16[]` pre-trigger data

Code	Function	Structure
51	**CAPTURE_DATA** A chunk of sampled data in a stream, block, or a triggered capture. More data will follow.	*Payload:* • `u8` stream serial number • `u16[]` sample data
52	**CAPTURE_END** Indicates the end of a multi-part capture. The payload may be empty if there is no more data to send (e.g. a stream had to be unexpectedly closed).	*Payload:* • `u8` stream serial number • `u16[]` sample data

12.11.3 ADC Commands

Code	Function	Structure
0	**READ_RAW** Get the last raw sample from enabled channels.	*Response:* • `u16[]` raw values 0–4095
1	**READ_SMOOTHED** Get the averaged values from enabled channels. Not available for high sample rates and when disabled.	*Response:* • `float32[]` smoothed values 0–4095
2	**READ_CAL_CONSTANTS** Read factory calibration constants from the MCU's ROM	*Response:* • `u16` VREFINT_CAL (raw word) • `u16` VREFINT_CAL_VADCREF (mV) • `u16` TSENSE_CAL1 (raw word) • `u16` TSENSE_CAL2 (raw word) • `u16` TSENSE_CAL1_TEMP (°C) • `u16` TSENSE_CAL2_TEMP (°C) • `u16` TSENSE_CAL_VADCREF (mV)
10	**GET_ENABLED_CHANNELS** Get numbers of all enabled channels (0-based)	*Response:* • `u8[]` enabled channel numbers
11	**GET_SAMPLE_RATE** Get the current sample rate (in Hz)	*Response:* • `u32` requested sample rate • `float32` real sample rate

Code	Function	Structure
20	**SETUP_TRIGGER** Configure the triggering level and other trigger parameters. This command does *not* arm the trigger!	*Request:* • `u8` source channel number • `u16` triggering level • `u8` active edge (1-falling, 2-rising, 3-any) • `u32` pre-trigger sample count • `u32` post-trigger sample count • `u16` hold-off time (ms) • `u8` auto re-arm (0,1)
21	**ARM** Arm the trigger for capture.	*Request:* • `u8` auto re-arm (0, 1, 255-no change)
22	**DISARM** Dis-arm the trigger.	
23	**ABORT** Abort any ongoing capture and dis-arm the trigger.	
24	**FORCE_TRIGGER** Manually trip the trigger, as if the threshold level was reached.	
25	**BLOCK_CAPTURE** Capture a fixed-length sequence of samples.	*Request:* • `u32` number of samples
26	**STREAM_START** Start a real-time stream of samples	
27	**STREAM_STOP** Stop an ongoing stream	
28	**SET_SMOOTHING_FACTOR** Set the smoothing factor ($\times 10^3$).	*Request:* • `u16` smoothing factor 0-1000
29	**SET_SAMPLE_RATE** Set the sampling frequency.	*Request:* • `u32` frequency in Hz
30	**ENABLE_CHANNELS** Select channels to sample. The channels must be configured in the unit settings.	*Request:* • `u32` bit map of channels to enable
31	**SET_SAMPLE_TIME** Set the sample time of the ADC's sample&hold circuit.	*Request:* • `u8` sample time 0–7

12.12 DAC Unit

The digital/analog unit works with the two-channel DAC peripheral of the microcontroller. It can be used in two modes: DC output, and waveform generation.

The waveform mode implements direct digital synthesis (explained in section 8.3.2) of a sine, rectangle, sawtooth or triangle wave. The generated frequency can be set with a sub-hertz precision up to the lower tens of kHz. The two outputs can use a different waveform shape, be synchronized, and their phase offset, as well as frequency, is dynamically adjustable.

12.12.1 DAC Configuration

```
[DAC:dac@13]
# Enabled channels (1:A4, 2:A5)
ch1_enable=Y
ch2_enable=Y
# Enable output buffer
ch1_buff=Y
ch2_buff=Y
# Superimposed noise type (NONE,WHITE,TRIANGLE) and nbr. of bits (1-12)
ch1_noise=NONE
ch1_noise-level=3
ch2_noise=NONE
ch2_noise-level=3
```

12.12.2 DAC Commands

Channels are specified in all commands as bit map:

- 0x01 - channel 1
- 0x02 - channel 2
- 0x03 - both channels affected at once

Code	Function	Structure
0	**WAVE_DC** Set a DC level, disable DDS for the channel	*Request:* • `u8` channels • `u16` level (0–4095)
1	**WAVE_SINE** Start a DDS sine waveform	*Request:* • `u8` channels
2	**WAVE_TRIANGLE** Start a symmetrical triangle waveform	*Request:* • `u8` channels

Code	Function	Structure
3	**WAVE_SAWTOOTH_UP** Start a rising sawtooth waveform	*Request:* • `u8` channels
4	**WAVE_SAWTOOTH_DOWN** Start a dalling sawtooth waveform	*Request:* • `u8` channels
5	**WAVE_RECTANGLE** Start a rectangle waveform	*Request:* • `u8` channels • `u16` on-time (0–8191) • `u16` high level (0–4095) • `u16` low level (0–4095)
10	**SYNC** Synchronize the two channels. The phase accumulator is reset to zero.	
20	**SET_FREQUENCY** Set the channel frequency	*Request:* • `u8` channels • `float32` frequency
21	**SET_PHASE** Set a channel's phase. It's recommended to set the phase only of one channel, leaving the other at 0°.	*Request:* • `u8` channels • `u16` phase (0–8191)
22	**SET_DITHER** Control the dithering function of the DAC block. A high noise amplitude can cause an overflow to the other end of the output range due to a bug in the DAC peripheral. Use value 255 to leave the parameter unchanged.	*Request:* • `u8` channels • `u8` noise type (0-none, 1-white, 2-triangle) • `u8` number of noise bits (1–12)

12.13 PWM Unit

The PWM unit uses a timer/counter to generate a PWM signal. There are four outputs with a common frequency and phase, but independent duty cycles. Each channel can be individually enabled or disabled.

This unit is intended for applications like light dimming, heater regulation, or the control of H-bridges.

12.13.1 PWM Configuration

```
[PWMDIM:pwm@12]
# Default pulse frequency (Hz)
frequency=1000
# Pin mapping - 0=disabled
# Channel1 - 1:PA6, 2:PB4, 3:PC6
ch1_pin=1
# Channel2 - 1:PA7, 2:PB5, 3:PC7
ch2_pin=0
# Channel3 - 1:PB0, 2:PC8
ch3_pin=0
# Channel4 - 1:PB1, 2:PC9
ch4_pin=0
```

12.13.2 PWM Commands

Code	Function	Structure
0	**SET_FREQUENCY** Set the PWM frequency	*Request:* • `u32` frequency in Hz
1	**SET_DUTY** Set the duty cycle of one or more channels	*Request:* • Repeat 1–4 times: – `u8` channel number 0–3 – `u16` duty cycle 0–1000
2	**STOP** Stop the hardware timer. Outputs enter low level.	
3	**START** Start the hardware timer.	

12.14 Touch Sense Unit

The touch sensing unit provides an access to the touch sensing controller. Its function is explained in section 8.4. The unit configures the TSC and reads the output values of each enabled touch pad.

12.14.1 Touch Sense Configuration

```
[TOUCH:touch@11]
# Pulse generator clock prescaller (1,2,4,...,128)
pg-clock-prediv=32
```

```
# Sense pad charging time (1-16)
charge-time=2
# Charge transfer time (1-16)
drain-time=2
# Measurement timeout (1-7)
sense-timeout=7

# Spread spectrum max deviation (0-128,0=off)
ss-deviation=0
# Spreading clock prescaller (1,2)
ss-clock-prediv=1

# Optimize for interlaced pads (individual sampling with others floating)
interlaced-pads=N

# Button mode debounce (ms) and release hysteresis (lsb)
btn-debounce=20
btn-hysteresis=10

# Each used group must have 1 sampling capacitor and 1-3 channels.
# Channels are numbered 1,2,3,4

# Group1 - 1:A0, 2:A1, 3:A2, 4:A3
g1_cap=
g1_ch=
# Group2 - 1:A4, 2:A5, 3:A6, 4:A7
g2_cap=
g2_ch=
# ...
```

12.14.2 Touch Sense Events

Code	Function	Structure
0	**BUTTON_CHANGE** The binary state of some of the capacitive pads with button mode enabled changed.	*Payload:* • u32 binary state of all channels • u32 changed / trigger-generating channels

12.14.3 Touch Sense Commands

Code	Function	Structure
0	**READ** Read the raw touch pad values (lower indicates higher capacitance). Values are ordered by group and channel.	*Request:* • `u16[]` raw values
1	**SET_BIN_THR** Set the button mode thresholds for all channels. Value 0 disables the button mode for a channel.	*Request:* • `u16[]` thresholds
2	**DISABLE_ALL_REPORTS** Set thresholds to 0, disabling the button mode for all pads.	
3	**SET_DEBOUNCE_TIME** Set the button mode debounce time (used for all pads with button mode enabled).	*Request:* • `u16` debounce time (ms)
4	**SET_HYSTERESIS** Set the button mode hysteresis.	*Request:* • `u16` hystheresis

Appendices