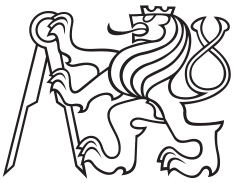
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering

Department of Measurement

Learning and automation GPIO platform

Bc. 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, 25. května 2018

.....

Acknowledgements

TODO

Abstract

This thesis documents the development of a general-purpose software and hardware platform for the interfacing of low-level hardware from high-level programming languages and applications run on the 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 eliminates the need for professional measurement and testing equipment.

Two hardware prototypes were designed and realized, accompanied by control libraries for programming languages C and Python. The Python library additionally integrates with MATLAB scripts. The devices provide access to hardware buses (I^2C, SPI, USART, 1-Wire) and microcontroller peripherals (ADC, DAC), implement frequency measurement and other useful features. The device is parametrised by a configuration file on a virtual disk accessible through USB, or written programmatically.

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

Figures

Tables

Acronyms

AC alternating current
ACM Abstract Control Model
ADC Analog/Digital Converter
API application programming interface

BFSK binary frequency-shift keying
BOT Bulk Only Transport

CAN Controller Area Network
CDC Communication Devices Class
CDC/ACM Communication Devices Class / Abstract Control Model
CPHA clock phase
CPOL clock polarity
CRC cyclic redundancy check
CSB Chip Select Bar
CTS Clear To Send

DAC Digital/Analog Converter
DALI Digital Addressable Lighting Interface
DC direct current
DDS Direct Digital Synthesis
DE Driver Enable
DFU Device Firmware Update
DMA Direct Memory Access
DTR Data Terminal Ready

FAT File Allocation Table
FS file system
FSK frequency-shift keying

GFSK Gaussian frequency-shift keying
GMSK Gaussian minimum-shift keying
GND ground
GPIO general purpose input/output
GPS Global Positioning System
GSM Global System for Mobile communications
GUI graphical user interface

HART Highway Addressable Remote Transducer

I²C Inter-Integrated Circuit
I²S Inter-IC Sound
IAD Interface Association Descriptor
IC integrated circuit
IDE integrated development environment
IRQ interrupt request
ISR interrupt service routine

LCD liquid crystal display
LED light emitting diode
LFN Long File Name
LIN Local Interconnect Network

MBR master boot record
M-Bus Meter Bus
MCU microcontroller unit
MISO Master In, Slave Out
MOSI Master Out, Slave In
MSC Mass Storage Class
MSK minimum-shift keying

NCO numerically controlled oscillator
NDIR nondispersive infrared
NFC near-field communication
NRZI Non Return to Zero Inverted
NSS Negated Slave Select
NVIC Nested Vectored Interrupt Controller

OOK on-off keying
OS operating system

PC personal computer
PCB printed circuit board
PMBus Power Management Bus
PWM pulse width modulation

RAM random-access memory
ROM read-only memory
RTC real-time clock
RTS Ready To Send

SAR successive approximation register
SCCB Serial Camera Control Bus
SCK Serial Clock
SCL Serial Clock Line
SCSI Small Computer System Interface
SDA Serial Data Line
SMBus System Management Bus
SPI Serial Peripheral Interconnect
SS Slave Select
SSH Secure Shell

STEM Science, Technology, Engineering and Mathematics

TCO temperature-compensated oscillator
TSC Touch Sensing Controller
TTL transistor-transistor logic
TVS transiet-voltage suppressor

TWI Two-Wire Interface

UART Universal Asynchronous Receiver/Transmitter
USART Universal Synchronous/Asynchronous Receiver/Transmitter
USB Universal Serial Bus

VCO voltage-controlled oscillator

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.

Recently, a wide range of intelligent sensors became available thanks to the drive to miniaturization in the consumer electronics industry. Those devices often provide 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.

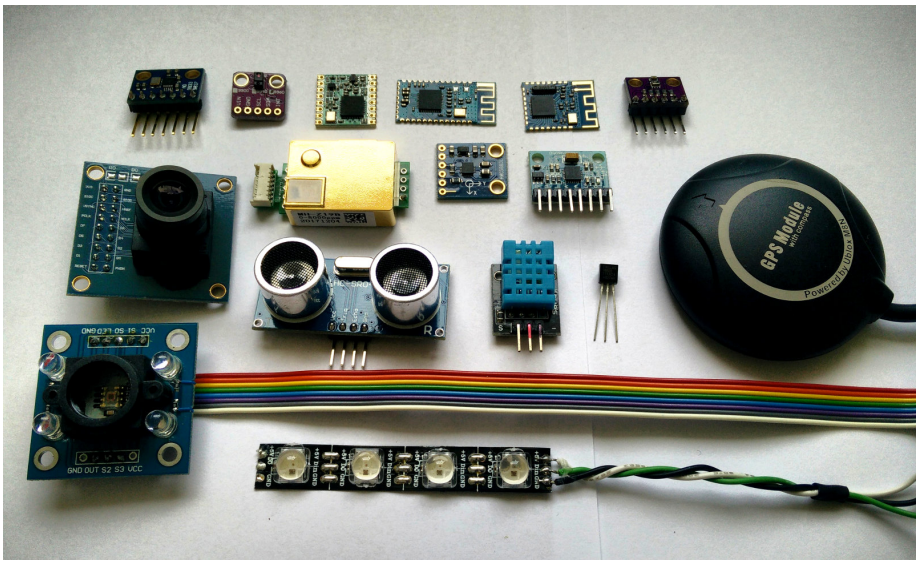


Figure 1.1: A collection of intelligent sensors and devices, most on breadboard adapters: (from the top left) a waveform generator, a gesture detector, a LoRa and two Bluetooth modules, an air quality and pressure sensor, a CO_2 sensor, a digital compass, an accelerometer, a GPS module, a camera, an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

If we wish to conduct experiments with those integrated modules, or just familiarize ourselves with a device before using it in a project, we need an easy way to interact with them. It would also be convenient to have direct access to low-level hardware, be it analog signal sampling, generation, or even just the access to logic inputs and outputs. However, advances in computer technology, namely the advent of the Universal Serial Bus (USB),

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 we want to perform measurements using a digital sensor, the usual route is to implement an embedded firmware for a microcontroller that connects to the personal computer (PC) through USB, or perhaps shows the results on a display. This approach has its advantages, but is time-consuming and requires specific knowledge unrelated to the measurements we wish to perform. It would be advantageous to have a way to access hardware without having to burden ourselves with the technicalities of this connection, even at the cost of lower performance compared to specialized devices or professional tools.

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 shall be, hereafter, called **GEX**, a name originating from "**G**PIO **Ex**pander".

1.1 Expected Outcome

It has been a long-time desire of the author to create a universal instrument connecting low-level hardware to a computer, and, with this project, it is finally being realized. Several related projects approaching this problem from different angles can be found on the internet; some of these will be presented in Chapter 3.

Our project is not meant to end with a tinkering tool that will be produced in a few prototypes and then forgotten. By creating 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.

Building on the experience with earlier embedded projects, an 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 printed circuit board (PCB). STM32 microcontrollers are relatively cheap and already popular in the embedded hardware community; there is a real possibility of the project gathering a community around it and growing beyond what will be presented in this paper.

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 of well known hardware buses have established themselves as the standard ways to interface digital sensors and modules: Serial Peripheral Interconnect (SPI), Inter-Integrated Circuit (I^2C) and Universal Synchronous/Asynchronous Receiver/Transmitter (USART) (UART in asynchronous mode) are some of the most common ones, often accompanied by a few extra general purpose input/output (GPIO) lines for features such as Reset, Chip Enable, or Interrupt. There are exceptions where silicon vendors have developed proprietary communication protocols that continue to be used either for historical reasons, or because of their specific advantages. An example is the Dallas Semiconductor 1-Wire bus used in digital thermometers.

Moving to industrial and automotive environments, we encounter various fieldbuses, Ethernet, Controller Area Network (CAN), current loop, Highway Addressable Remote Transducer (HART), Local Interconnect Network (LIN), Digital Addressable Lighting Interface (DALI), RS-485 (e.g., for Modbus), Meter Bus (M-Bus), PLC-BUS, and others. Those typically use transceiver integrated circuits (ICs) and other circuitry, such as transiet-voltage suppressors (TVSs), signal filters, or galvanic isolation. 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 is necessary to use a traditional analog sensor, capture a transient waveform, or to just measure voltage. GEX is meant to focus on digital interfaces, however giving it this capability makes it much more versatile. Nearly all microcontrollers include an Analog/Digital Converter (ADC) 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 is 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 have covered some more advanced features, but skipped the simplest feature: direct access to GPIO pins. Considering the latencies of USB and the PC's operating system (OS), this cannot be used reliably for "bit banging"; however, we can still accomplish a lot with just changing logic levels—e.g., to control character liquid crystal displays (LCDs), or emulate some interfaces that include a clock line, like SPI. As mentioned in Section 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 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 waveform generation.

2.2 Connection to the Host Computer

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 UART. 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, or through raw access to the corresponding USB endpoints. Using raw access avoids potential problems with the OS'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 good serial port support (such as National Instruments LabWindows CVI or MATLAB).

A connection using a hardware UART is also planned, as a fallback for boards without an USB connector or for platforms with no USB connectivity. A wireless connection to the host PC should also be possible and work transparently in a similar way to the USB or UART connection.

2.2.2 Configuration Files

The module must be easily reconfigurable. Given the settings are almost always going to be tied to 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 graphical user interface (GUI), instead using the built-in applications of the PC OS to view and edit files. Besides the configuration files, 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

Summarizing the preceding discussion, we obtain the following list of features to implement in the GEX firmware:

- **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, I²C, UART/USART, 1-Wire

- **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 interface 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 its low cost, advanced peripherals, and the availability of development boards. The firmware can be ported to other microcontroller units (MCUs) later (e.g., to STM32L072, STM32F103 or STM32F303).

The STM32F072 is an Arm Cortex-M device with 128 KiB of flash memory, 16 KiB of random-access memory (RAM) and running at 48 MHz. It is equipped with a USB Full Speed peripheral block, a 12-bit ADC and Digital/Analog Converter (DAC), a number of general-purpose timers/counters, SPI, I²C, 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.

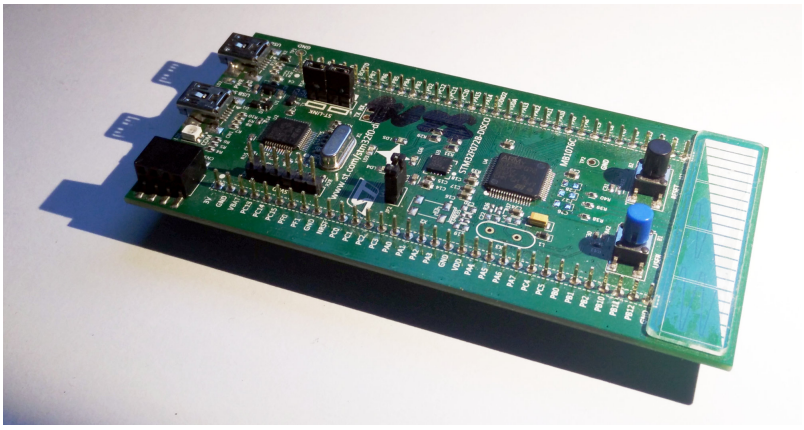


Figure 2.1: A Discovery development board with the STM32F072 microcontroller

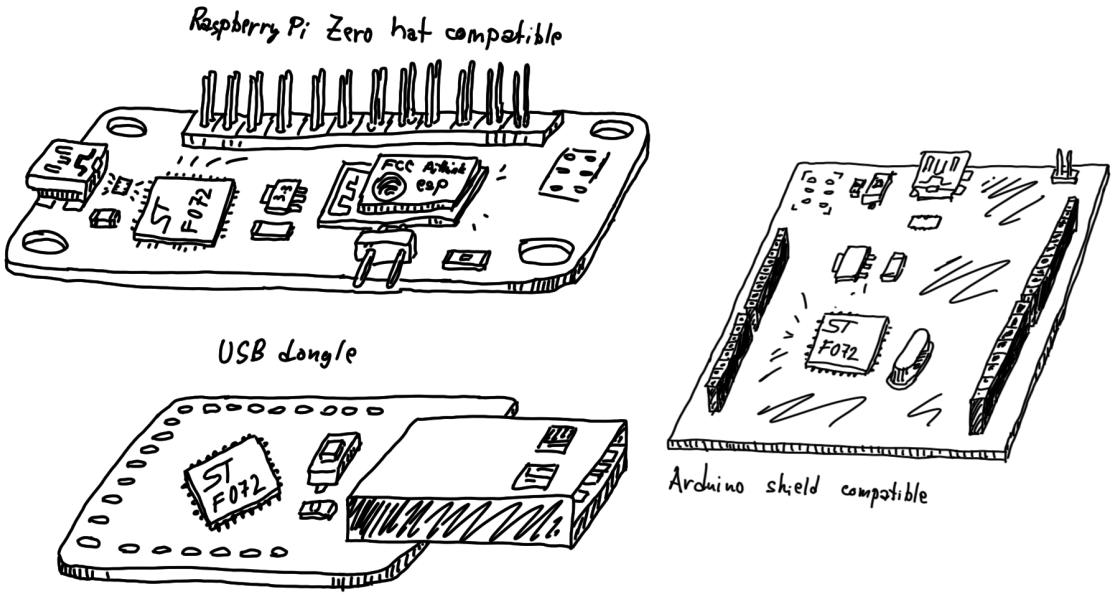


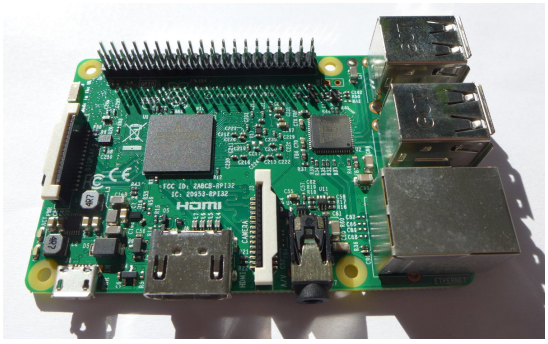
Figure 2.2: A sketch of three possible form factors for a GEX hardware realization

Chapter 3

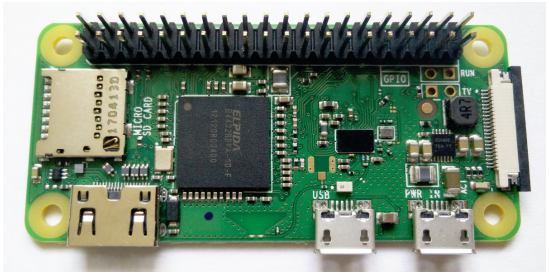
Existing Solutions

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 GPIO, SPI, I^2C, UART, and PWM, with other protocols and functions 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 Science, Technology, Engineering and Mathematics (STEM). 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 minicomputer, with an attached monitor and a keyboard, or use some form of remote access (e.g., Secure Shell (SSH), or screen sharing).

3.2 Bus Pirate

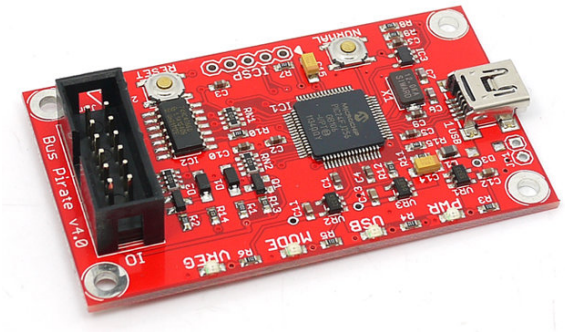


Figure 3.2: Bus Pirate v.4 (photo taken from [1])

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 ADC only has a resolution of 10 bits (1024 levels). There is no DAC available on the chip, which makes applications that require a varied output voltage more difficult to implement. 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 I^2C/SPI Interface Device which also includes several GPIO lines, their USB DAQ module, or some of the Total Phase I^2C/SPI gadgets (Figure 3.3).

The performance GEX can provide may not always match that of those professional tools, but in many cases it will be a sufficient substitute at a fraction of the cost.

(a) : NI I^{2}C/SPI Interface Device

(b) : NI USB DAQ module

(c) : Total Phase SPI/I^{2}C Host "Aardwark"

Figure 3.3: An example of professional tools that GEX could replace in less demanding scenarios *(pictures taken from marketing materials: [2, 3, 4])*

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 and powerful interface which replaces several older technologies; for this reason its specification is very complex and going into all details is hardly possible. We will cover the basic principles and terminology of USB and focus on the parts relevant for the GEX project. More information about the bus can be found in the official specification [5], related documents published by the USB Implementers Forum, and other on-line resources [6, 7].

4.1 Basic Principles and Terminology

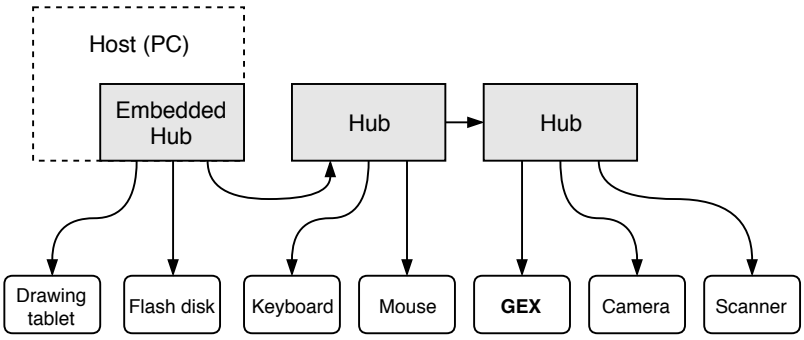


Figure 4.1: The hierarchical structure of the USB bus

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* [8].

4.1.1 Pipes and Endpoints

Communication between the host and a function is organized into virtual channels called *pipes* connecting to the device's *endpoints*, identified by endpoint numbers.

Endpoints can be either unidirectional or bidirectional; the direction from the host to a function is called OUT, the other direction (function to host) is called IN. A bidirectional endpoint is technically composed of IN and OUT endpoints 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

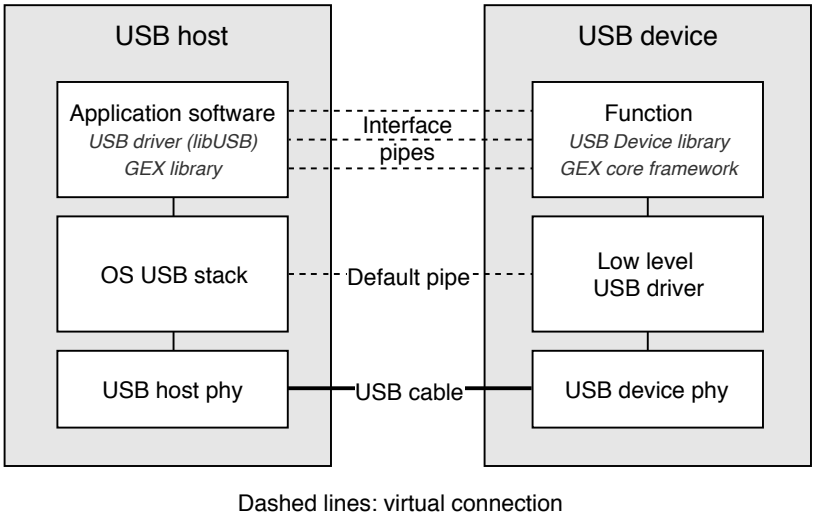


Figure 4.2: The logical structure of USB

host uses the control endpoint to read information about the device and configure it as needed.

4.1.2 Transfer Types

There are four types of data transfers defined in USB: 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 application-specific control messages that can affect other pipes.
- *Bulk* – used for burst transfers of large messages
- *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

4.1.3 Interfaces and Classes

The function's endpoints are grouped into *interfaces*. An interface describes a logical connection of endpoints, such as the reception and transmission endpoints that belong together. An interface is assigned a *class* defining how it should be used.

Standard classes are defined by the USB specification [9] 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.

4.1.4 Descriptors

USB devices are introspectable, that is, the host can learn about a newly connected device automatically by probing it, without any user interaction. This is accomplished using a *descriptor table*, a binary structure stored in the function and read by the host through the control endpoint (default pipe) after the device is attached.

Each descriptor starts with a declaration of its length (in bytes), followed by its type, allowing the host to skip unknown descriptors without having to discard the rest of the table. The descriptors are logically nested and form a tree-like structure, but they are stored sequentially in the descriptor table and the lengths do no include sub-descriptors.

The topmost descriptor holds information about the entire function, including the vendor and product IDs which uniquely identifies the device model. It is followed by a Configuration descriptor, grouping a set of interfaces. More than one configuration may be present and available for the host to choose from; however, this is rarely used or needed. Each configuration descriptor is followed by one or more interface descriptors, each with its class-specific sub-descriptors and/or endpoint descriptors.

The descriptor table used by GEX is captured in Figure 4.3 for illustration. The vendor and product IDs were obtained from the pid.codes repository [10] providing free product codes to open source projects. The official way of obtaining the unique code involves high recurring fees ($4000 per annum) to the USB Implementers Forum, Inc. and is therefore not affordable for non-commercial use; alternatively, a product code may be obtained from some MCU vendors if their product is used in the device.

```
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass          239 Miscellaneous Device
  bDeviceSubClass         2
  bDeviceProtocol         1 Interface Association
  bMaxPacketSize0        64
  idVendor           0x1209 InterBiometrics
  idProduct          0x4c60
  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`"

4.2 USB Physical Layer

USB uses differential signaling with Non Return to Zero Inverted (NRZI) encoding and bit stuffing (the insertion of dummy bits to prevent long intervals in the same direct current (DC) level). The encoding, together with frame formatting, checksum verification, retransmission, and other low-level aspects of the USB connection are entirely handled by the USB physical interface block in the microcontroller's silicon. Normally we do not need to worry about those details; nonetheless, a curious reader may find more information in chapters 7 and 8 of [5]. The electrical characteristics of the physical connection deserve more attention, as they need to be understood correctly for a successful schematic and PCB design.

The USB cable contains 4 conductors: V_{BUS} (+5 V), D+, D–, and ground (GND). The data lines, D+ and D–, are also commonly labeled DP and DM. This differential pair should be routed in parallel on the PCB and kept at the same length.

USB versions that share the same connector are backward compatible. The desired bus speed is requested by the device using a 1.5 kΩ pull-up resistor to 3.3 V on one of the data lines: D+ pulled high for Full Speed (shown in Figure 4.4), D– pulled high for Low Speed. The polarity of the differential signals is also inverted depending on the used speed, as the idle level changes. Some microcontrollers integrate the correct pull-up resistor inside the USB peripheral block (including out STM32F072), removing the need for an external resistor.

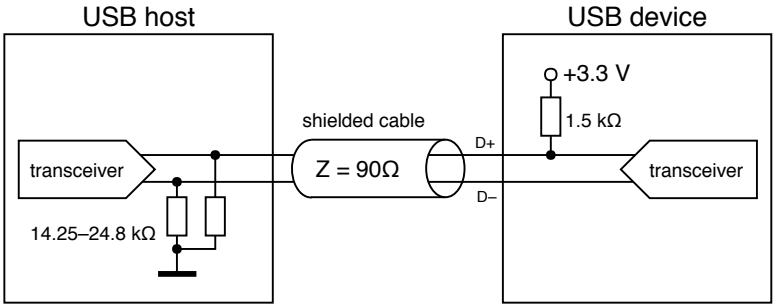


Figure 4.4: Pull-up and pull-down resistors near the host and a Full Speed function, as prescribed by the USB specification rev. 2.0

When a function needs to be re-enumerated by the host, which causes a reload of the descriptor table and the re-attachment of software drivers, it can momentarily remove the pull-up resistor, which the host will interpret as if the device was disconnected. With an internal pull-up, this can be done by flipping a bit in a control register. An external resistor may be connected through a transistor controlled by a GPIO pin. As discussed in [11], a GPIO pin might be used to drive the pull-up directly, though this has not been verified by the author.

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.

More details about the electrical and physical connection may be found in [6], sections *Connectors* through *Power*.

4.3 USB Classes

This section explains the classes used in the GEX firmware. A list of all standard classes with a more detailed explanation can be found in [9].

4.3.1 Mass Storage Class

The Mass Storage Class (MSC) is supported by all modern PC operating systems to support USB thumb drives, external disks, memory card readers, and other storage devices.

The MSC specification [12] defines multiple *transport protocols* that can be selected using the descriptors. The Bulk Only Transport (BOT) [13] will be used for its simplicity. BOT uses two bulk endpoints for reading and writing blocks of data and for the exchange of control commands and status messages.

For the mass storage device to be recognized by the host operating system, it must also implement a *command set*. Most mass storage devices use the *Small Computer System Interface (SCSI) Transparent command set* [1].

Unfortunately, the SCSI Transparent command set appears to have been deliberately left unspecified for license or copyright reasons (see discussion in [14] and the surrounding thread) and the protocol now used under this name is an industry standard without a clear definition. Some pointers may be found in [15] and by examining the source code of the USB Device driver library provided by ST Microelectronics.

This command set lets the host read information about the attached storage device, 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 ensures that all internal buffers have been written to the flash memory.

In order to emulate a mass storage device without having a physical storage medium, we need to generate and parse the file system 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, Communication Devices Class / Abstract Control Model (CDC/ACM) is now the de facto standard way of making USB devices appear as serial ports on the host OS. Its specification can be found in [16]. CDC/ACM is a combination of two related classes, CDC handling the data communication and ACM,

[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 [12]

which defines control commands. Three endpoints are used: bulk IN, bulk OUT, and interrupt OUT.

The interrupt endpoint is used for control commands, such as toggling the auxiliary lines of RS-232 or setting the baud rate. Since GEX does not translate the data communication to any physical UART, those commands are not applicable and can be silently ignored.

An interesting property of the CDC class is that the bulk endpoints transport raw data without any wrapping frames. By changing the interface's class in the descriptor table to 255 (*Vendor Specific Class*), we can retain the messaging functionality of the designated endpoints, while accessing the endpoints device directly using, e.g., libUSB, without any interference from the OS. This approach is also used to hide the MSC interface when it is not needed.

4.3.3 Interface Association: Composite Class

The original USB specification expected that each function will have only one interface enabled at a time. After it became apparent that there is a need to have multiple unrelated interfaces working in parallel, the Interface Association Descriptor (IAD) [17] was introduced as a workaround.

The 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 0xEF, subclass 0x02, and protocol 0x01 in the top level descriptor, so that the OS knows to look for this descriptor 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* was created as a wrapper for the library-provided MSC and CDC/ACM implementation.

Chapter 5

FreeRTOS

FreeRTOS is a free, open-source real-time operating system kernel targeted at embedded systems; it has been ported to many different microcontroller architectures [18] and it is the de-facto industry standard. The system is compact and designed to be easy to understand; it is written in C, with the exception of some architecture-specific routines which use assembly. A complete overview of its application programming interface (API) is available in the FreeRTOS reference manual [19] and its guide book [20].

FreeRTOS provides a task scheduler, forming the central part of the system, and implements queues, semaphores, and mutexes for message passing and the synchronization of concurrent tasks. Those features are summarily called *synchronization objects*, or simply *objects*.

The system is used in GEX for its synchronization objects that allow us to safely pass messages between interrupts and working threads, without deadlocks or race conditions; the particular usage of FreeRTOS features will be explained in Section 10.7. The built-in stack overflow protection helps us optimize task memory allocation[1], and the heap allocator provided by FreeRTOS enables thread-safe dynamic allocation with a shared heap.

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 holding structure with its name, saved *context*, and other metadata used by the kernel. A task 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 the stack before leaving an interrupt service routine (ISR). The FreeRTOS website provides an example with the AVR port [21] demonstrating how its internal functionality is implemented, including the context switch.

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, always giving a task one tick of run time (usually 1 ms) before

[1] The stack monitor reports how much stack space was really used, so we can expand or shrink it as needed to make the application work reliably, without wasting memory that would never be used.

interrupting it. Which task should run is determined primarily by their priority numbers, but there are other factors, as will be shown in Section 5.1.1.

Task Run States

Tasks can be in one of four states: Suspended, Ready, Blocked, and Running. The Suspended state does not normally occur in a task's life cycle, it is entered and left using API calls from the application. 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 a loop until another task is available. The Idle task is always either Ready or Running and has the lowest priority of all tasks.

Task Switching and Interrupts

Task switching occurs periodically in a timer interrupt, usually every 1 ms; in Arm Cortex-M chips this is typically the SysTick interrupt, a timer designed for this purpose that is included in the core itself and thus available on all derived platforms.

After one tick of run time, the Running task is paused and becomes Ready, or continues to run if no higher-priority task is available. If a higher-priority task waits for an object and this is made available in an ISR, the running lower-priority task is paused and the waiting task resumes immediately. FreeRTOS defines interrupt-friendly variants of some of the API functions intended for this purpose; however, only a subset of the API is available in an ISR, for example, it is not possible to use the delay function or wait for an object with a timeout, as the SysTick interrupt, incrementing the tick counter, has the lowest priority and could not run. This is by design, intended to prevent unexpected context switching in application 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 earlier 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, which will now be briefly explained; a more in-depth description can be found in the guide book [20].

- **Binary semaphores** are used for task notifications, e.g., when a task waits for a semaphore to be set by an ISR. This makes the task Ready and if it has a higher priority than the task previously running, it is immediately resumed to process the event.

- **Counting semaphores** represent available resources in a resource pool, a set of software or hardware resources used by tasks. The pool is accompanied by a counting semaphore on which tasks wait for a resource to become available, and then subtract the semaphore value. After a resource is no longer needed by the task, the semaphore is incremented again and another task can use it.
- **Mutexes** (locks) are similar to semaphores, but they must be taken and released in the same task. We use them to guard an exclusive access to a resource, typically a hardware peripheral or a shared memory area. When a mutex is taken, any other tasks trying to take it too enter become Blocked. A Blocked task waiting for a mutex is resumed once this becomes available, at which point the task becomes its owner and is resumed.
- **Queues** are used for passing messages between tasks, or from interrupts to tasks. Both sending and receiving of queue messages can block the task until the operation becomes possible. A queue handing task is often simply a loop which tries to read from the queue with an infinite timeout and processes the received data once the reading succeeds.

It must be noted that synchronization objects like mutexes and semaphores can help combat concurrent access only when used consistently and correctly. A locked mutex cannot guard against a rogue task accessing the protected resource without checking.

5.2 Stack Overflow Protection

Each task in FreeRTOS is assigned a block of RAM to use as its stack when it runs. This is where the stack pointer is restored to in the context switch. The stack pointer could move outside the designated area if the allocated space is insufficient; without countermeasures, this would mean that we are overwriting bytes in some unrelated memory structure, perhaps another task's stack memory.

A stack overflow protection can be enabled by a flag in the FreeRTOS configuration file. This function works in two ways: the more obvious is a simple check that the stack pointer remains in the designated area; however, as the check may be performed only in the scheduler interrupt, it is possible that the stack pointer exceeds the bounds only temporarily and returns back before the check can run. A more advanced solution, used by FreeRTOS, fills the stack memory with a known filler value before starting the task; the last few bytes are then tested to match this value. Not only can we detect a stack overflow more reliably, this feature also makes it possible to estimate the peak stack usage by counting the remaining filler bytes. We cannot distinguish between the original values and the same data stored on the stack by the program, but the possibility of this happening is sufficiently low and this method proves remarkably successful at detecting misconfigured stack sizes.

Chapter 6

The FAT16 File System and Its Emulation

A file system (FS) is used by GEX to provide the user 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 good cross-platform support [22].

Three variants of the File Allocation Table (FAT) file system exist: FAT12, FAT16, and FAT32. FAT12 was used on floppy disks and 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 could not support their larger capacity.

This chapter will explain the structure of FAT16 and the challenges faced when trying to emulate it without a physical storage medium. A more detailed overview of the file system can be found in literature [23, 24, 25, 26, 27] consulted during the GEX firmware development, with the Microsoft white paper [27] giving the most complete description.

6.1 The General Structure of the FAT File System

The storage medium is organized into *sectors* (or *blocks*), usually 512 bytes long; that is the smallest addressing unit used by the file system. The disk starts with a *boot sector*, also called the master boot record (MBR), 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 either of [23, 24, 25, 26, 27] or in the attached GEX source code.

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 special value:

- 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.

6.1.3 Root Directory

A directory is a record on the disk that can span several clusters and holds information about the files and sub-directories contained in it. The root directory has the same structure as any other directory; the difference lies in the fact that it is allocated with a fixed position and size when the disk is formatted, whereas other directories are stored in the same way as ordinary files and their capacity can be increased by simply expanding to another cluster.

Directories are organized in 32-byte entries representing individual files. Table 6.2 shows the structure of one such entry. The name and extension fields form the well-known "8.3" file name format known from MS DOS[1]. Longer file names are encoded using the Long File Name (LFN) scheme [28] as special hidden entries stored in the directory table alongside the regular "8.3" ones kept for backward compatibility.

The first byte of the file name has special meaning:

- 0x00 – indicates that there are no more files when searching the directory

[1] "8.3" refers to the byte size of the name and extension fields in the directory entry.

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

- 0xE5 – marks a free slot; this is used when a file is deleted
- 0x05 – indicates that the first byte should actually be 0xE5, a code used in some character sets at the time, and the slot is *not* free[2].
- Any other values, 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 defining the disk's label shown by the host OS. A file with the directory bit set is actually a pointer to a subdirectory, meaning that when we open the linked cluster, we will find another 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; the exact sizes are stored in the files' directory entries.

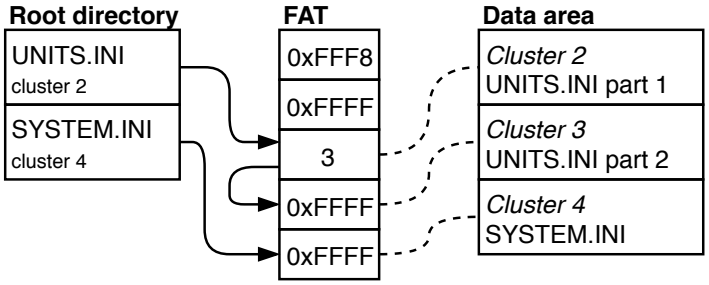


Figure 6.1: An example of the GEX virtual file system

[2]The special meaning of 0xE5 appears to be a correction of a less than ideal design choice earlier in the development of the file system

6.2 FAT16 Emulation

The FAT16 file system is relatively straightforward to implement. However, it is not practical or even possible to keep the entire file system in memory on a small microcontroller like our STM32F072. 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 MSC and provides API endpoints to which we connect our file system emulator. Specifically, those are requests to read and write a sector, and to the read disk's status and its parameters, such as the capacity.

6.2.1 DAPLink Emulator

A FAT16 emulator was developed as part of the open-source Arm Mbed DAPLink project [29]. It is used there for a drag-and-drop flashing of firmware images to the target microcontroller, taking advantage of the inherent cross-platform support (it uses the same software driver as any thumb drive, as discussed in Section 4.3.1). Arm Mbed also uses a browser-based integrated development environment (IDE) and cloud build servers, thus the end user does not need to install or set up any software to program a compatible development kit.

The GEX firmware adapts several parts of the DAPLink code, optimizing its RAM usage and porting it to work with FreeRTOS. The emulator source code is located in the `User/vfs` folder of the GEX repository; the original Apache 2.0 open-source software license headers, as well as the file names, have been retained.

As shown in Table 6.1, the disk consists of several areas. The boot sector is immutable and can be loaded from the flash memory when requested. The handling of the other disk areas (FAT, data area) depends on the type of access: read or write.

6.2.2 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 dynamically generated from the binary settings storage and, conversely, parsed as a byte stream without ever existing in their full form. This fact makes our task more challenging, as the file size cannot be easily measured and there is 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 generation routine: a *read window* and a *dummy read mode*.

A read window is a specification of the byte range we wish to generate. The INI generator discards bytes before the start of the read window, writes those inside the window to a holding buffer, and stops the end of the window 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 is never reached (e.g., 0xFFFFFFFF), and have the generator count discarded characters. This character counter holds the full file size once the generation is completed.

One more problem needs to be addressed: we need to know the mapping between the files and the clusters they are stored in. In our case, the files change only when the settings

are modified. After each such change, an algorithm is run which measures the file sizes, allocates their clusters, and preserves this information for later use. When the host tries to read from the data area of the disk, we simply test if the requested sectors are occupied by any file, and if so, serve the corresponding part of it using the read window function. The FAT can be dynamically generated from this information as well.

6.2.3 Write Access

Write access to the disk is more challenging to emulate than reading, as the host OS tends to be somewhat unpredictable. In GEX's case we are interested only in the action of overwriting an already existing file, but it is interesting to analyze other actions the host may perform as well.

It must be noted that due to the nonexistence of a physical storage medium, it is not possible to read back a file the host has previously written, unless we store or re-generate its content when such a read attempt occurs. The OS may show the written file on the disk, but when the user tried to open it, the action either fails, or shows a cached copy. The emulator works around this problem by temporarily reporting that the storage medium has been removed after a file is written, forcing the host to drop any cached data and reload the disk.

File Deletion

A file is deleted by:

1. Marking all FAT sectors used by the file as free
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 entire written sector as a directory table and compare it with our knowledge of its original contents.

6.2.4 File Name Change

A file is renamed by modifying its directory entry. In the simple case of a short, 8.3 file name, this is an in-place modification of the file entry. Long file names, using the LFN extension, are a complication, as the number of non-file entries holding the long file name might change, and subsequently the following entries in the table may shift or be re-arranged.

6.2.5 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

We can expect that the host first finds available sectors and a free directory entry before performing any write operations, to prevent potential disk corruption.

To properly handle a newly created 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, the two latter steps may happen in any order, and often the content is written before the directory table is updated.

The uncertain order of the written areas poses a problem when the file name has any significance, as we cannot store the received file data while waiting for the directory table to be updated. The Arm DAPLink firmware solves this by analyzing the content of the first written sector of the file, which may contain the binary Nested Vectored Interrupt Controller (NVIC) table, or a character pattern typical for Intel hex files, allowing it to recognize a binary image the user wants to flash to the target MCU.

6.2.6 File Content Change

A change to 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 unknown until the content is written, but we could detect the file name by comparing the start 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 is also no guarantee regarding the order in which the file's sectors are written. A non-linear or partial file update is hard to process for the emulator, but it can be reliably detected and discarded. Fortunately, this host behavior has not been conclusively observed in practice, but a file update rarely fails for unknown reasons; 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 details.

7.1 UART and USART

The Universal Synchronous/Asynchronous Receiver/Transmitter (USART) 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. RS-232 can be considered the ancestor of USB in its widespread availability and use. UART framing is also used in the industrial bus RS-485 and the automotive interconnect bus LIN.

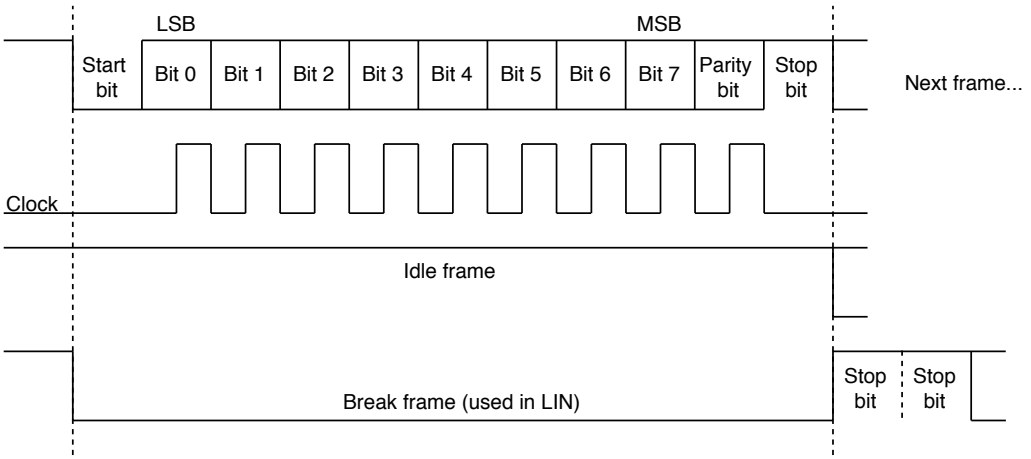


Figure 7.1: USART frame format in the 8-bit configuration with parity

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 USART frame, shown 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), dividing consecutive frames.

RS-232 uses the UART framing, but its levels are different: logical 1 is represented by negative voltages −3 to −25 V and logical 0 uses the same range, but positive. To convert between RS-232 levels and transistor-transistor logic (TTL) (5 V) levels, a level-shifting circuit such as the MAX232 can be used. In RS-232, the two data lines (Rx and Tx) are accompanied by Ready To Send (RTS), Clear To Send (CTS), and Data Terminal Ready (DTR), which facilitate handshaking and hardware flow control. In practice, those additional signals are often unused or their function differs from their historical meaning; 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 [30].

7.1.1 Examples of Devices Using UART

- **MH-Z19B** – nondispersive infrared (NDIR) CO_2 concentration sensor
- **NEO-M8** – uBlox Global Positioning System (GPS) module
- **ESP8266** with AT firmware – a WiFi module
- **MFRC522** – near-field communication (NFC) MIFARE reader/writer IC (also supports other interfaces)

7.2 SPI

Serial Peripheral Interconnect (SPI) 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 7.3. It uses at least 4 wires: Serial Clock (SCK), Master Out, Slave In (MOSI), Master In, Slave Out (MISO) and Slave Select (SS). SS is often marked Chip Select Bar (CSB) or Negated Slave Select (NSS) to indicate that its active state is 0. Slave devices are addressed using their SS input while the data connections are shared. A slave that is not addressed releases the MISO line to a high impedance state so it does not 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. The slave normally responds with 0x00 or a status register as the first byte of the response, before it can process the received command. A timing diagram is shown in Figure 7.2, including two configurable parameters: clock polarity (CPOL) and clock phase (CPHA).

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 is a read or write access, and an address field selecting the target register. The slave then either stores the following MOSI byte(s) into the register, or sends its content back on MISO (or both simultaneously).

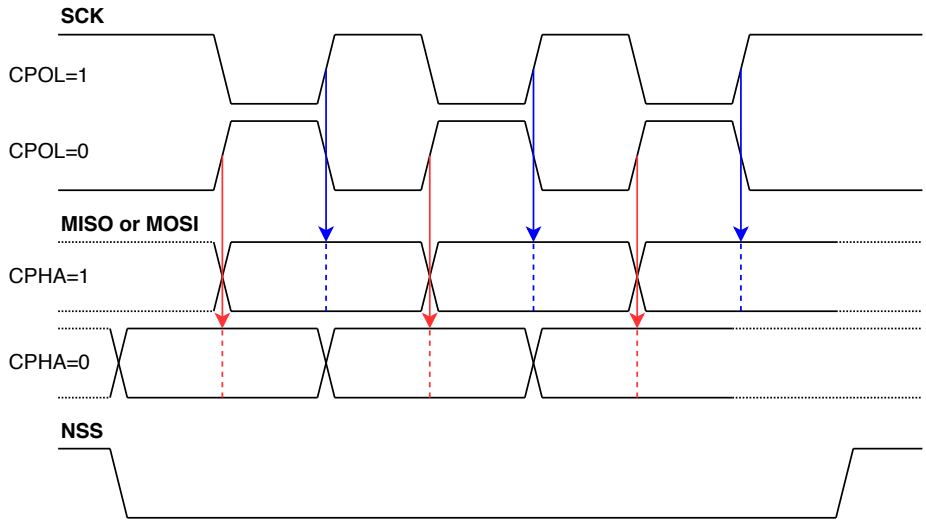


Figure 7.2: SPI timing diagram explaining the CPOL and CPHA settings (shown on 3 data bits; a real message will use at least 8 bits)

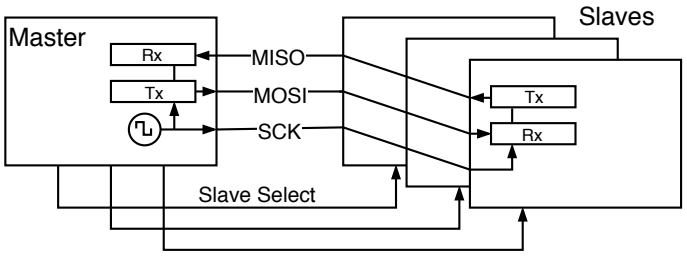


Figure 7.3: A SPI bus with 1 master and 3 slaves, each enabled by its own Slave Select signal

7.2.1 Examples of Devices Using SPI

- **SX1276** – LoRa transceiver
- **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** – waveform generator (MOSI only)
- **ADE7912** – triple Σ-Δ ADC for power metering applications
- **SD cards** [31]
- SPI-interfaced EEPROM and Flash memories

7.3 I²C

Inter-Integrated Circuit (I²C) is a two-wire, open-drain bus that supports multi-master operation. It uses two connections (plus GND): Serial Data Line (SDA) and Serial Clock Line (SCL), both open-drain with a pull-up resistor.

The protocol was developed by Philips Semiconductor (now NXP Semiconductors), and its implementors were, until 2006, 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 real-time clock (RTC) chip). I²C is a basis of the System Management Bus (SMBus) and Power Management Bus (PMBus), which add additional constraints and rules for a more robust operation.

The frame format is shown and explained in Figure 7.4; more details may be found in the specification [32] or application notes and datasheets offered by chip vendors, such as the white paper from Texas Instruments [33]. A frame starts with a start condition and stops with a stop condition, defined by an SDA edge while the 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. Slow slave devices may stop the master from sending more data by holding the SCL line low at the end of a byte, a feature called *Clock Stretching*. As the bus is open-drain, the line cannot go high until all participants release it.

Two addressing modes are defined: 7-bit and 10-bit. Due to the small address space, exacerbated by many devices implementing only the 7-bit addressing, collisions between different chips on a shared bus 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.

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 [32]. This feature is, however, rarely used in practice; the most common topology for I²C is multi-drop single-master, similar to SPI, with the advantage of using only two microcontroller pins.

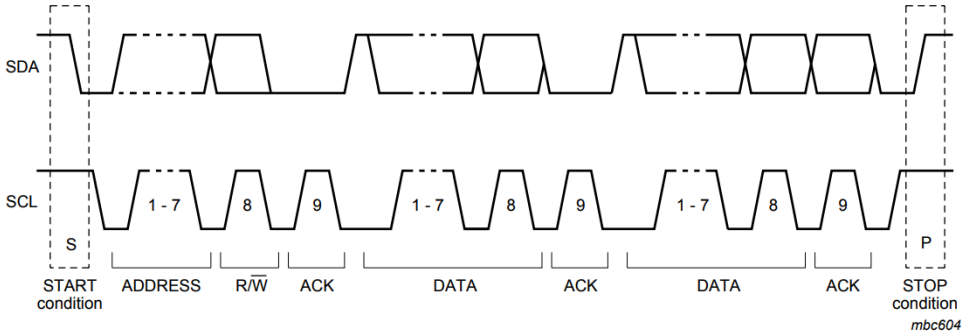


Figure 7.4: An I²C message diagram (*taken from the I²C specification [32]*)

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** – a light emitting diode (LED) matrix driver
- The Serial Camera Control Bus (SCCB) used to configure camera modules is derived from I²C

7.4 1-Wire

The 1-Wire bus, developed by Dallas Semiconductor (acquired by Maxim Integrated), uses a single, bi-directional data line, which can also power the slave devices in a *parasitic mode*, reducing the number of required wires to just two (compare with 3 in I²C and 5 in SPI, all including GND). The parasitic operation is possible thanks to the data line resting at a high logic level most of the time, charging an internal capacitor.

1-Wire uses an open-drain connection for the data line, similar to I²C, though the protocol demands it to be connected directly to V_dd in some places when the parasitic mode is used; this is accomplished using an external transistor, or by reconfiguring the GPIO pin as output and setting it to 1, provided the microcontroller is able to supply a sufficient current.

The communication consists of short pulses sent by the master and (for bit reading) the line continuing to be held low by the slave for a defined amount of time. The pulse timing determines whether it is a read or write operation and which value is encoded. It can be implemented either in software as delay loops, or by abusing a UART peripheral, as explained in [34]. Detailed timing diagrams can be found in the DS18x20 [35]. 1-Wire transactions include a checksum byte to ensure an error-free communication.

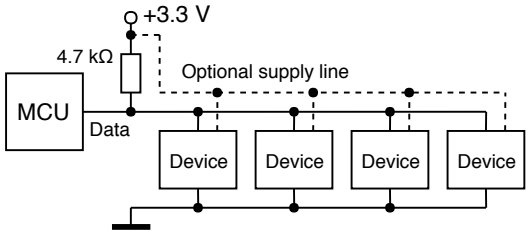


Figure 7.5: 1-Wire connection topology with four slave devices

Devices are addressed by their unique 64-bit ID numbers called ROM codes or ROMs; they can be found by the bus master, with a cooperation from slaves, using a ROM Search algorithm. The search algorithm is explained in [36], including a possible implementation example. If only one device is connected, a wild card command Skip ROM can be used to address the device without a known ROM code.

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.

7.5 NeoPixel

NeoPixel is a marketing name of the **WS2812** and compatible intelligent LED drivers that are commonly used in "addressable LED strips". Additional technical details about the chips and their protocol may be found in the WS2812B datasheet [37]. These chips include the control logic, PWM drivers and the LED diodes all in one 5×5 mm SMD package.

The NeoPixel protocol is unidirectional, using only one data pin. The LED drivers are chained together. Ones and zeros are encoded by pulses of a defined length on the data pin; after the color data was loaded into the LED string, a longer "reset" pulse (low level) is issued by the bus master and the set colors are displayed. The timing constraints are listed in Table 7.1.

The NeoPixel timing is sensitive to pulse length accuracy; a deviation from the specified timing may cause the data to be misinterpreted by the drivers. Some ways to implement the timing use hardware timers or the Inter-IC Sound (I^2S) peripheral. An easier method that does not require any additional hardware resources beyond the GPIO pin is to implement the timing using delay loops in the firmware; care must be taken to disable interrupts in the sensitive parts of the timing; it may be advantageous to implement it in assembly for a tighter control.

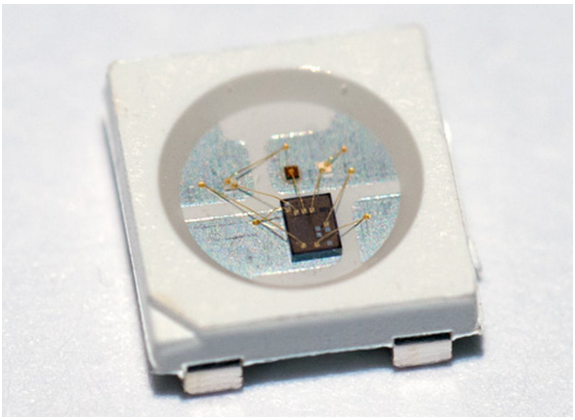


Figure 7.6: A close-up photo of a WS2812B pixel, showing the LED driver IC

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

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 voltage-controlled oscillator (VCO) 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 can be realized using a timer/counter peripheral of a microcontroller.

Two basic methods to measure frequency exist [38], each with its advantages and drawbacks:

- The *direct method* (Figure 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. Further increase of precision is possible through analog or digital interpolation [39], a method used in some professional equipment.

- The *indirect* or *reciprocal method* (Figure 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.

  The 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 prescaler[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.

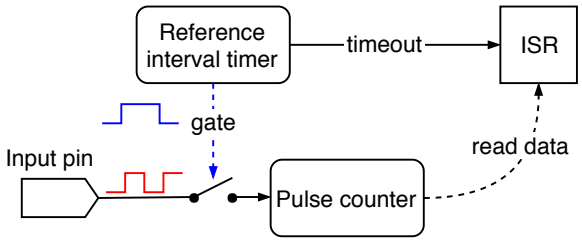


Figure 8.1: Direct frequency measurement method

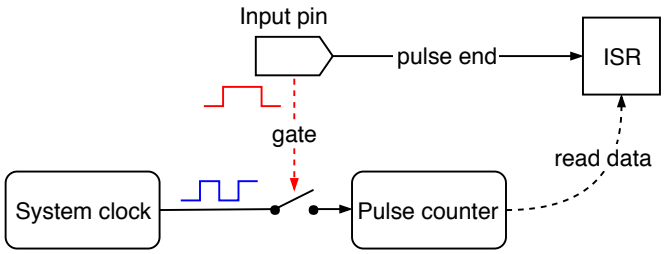


Figure 8.2: Reciprocal frequency measurement method

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, for cases where we do not know the expected frequency beforehand, could be to obtain an estimate using the direct method first, and if the frequency is below the worst-case error crossing point (here 7 kHz, according to Figure 8.3), 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, with the synchronization pulse provided by a GPS receiver to time the measurement interval.

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

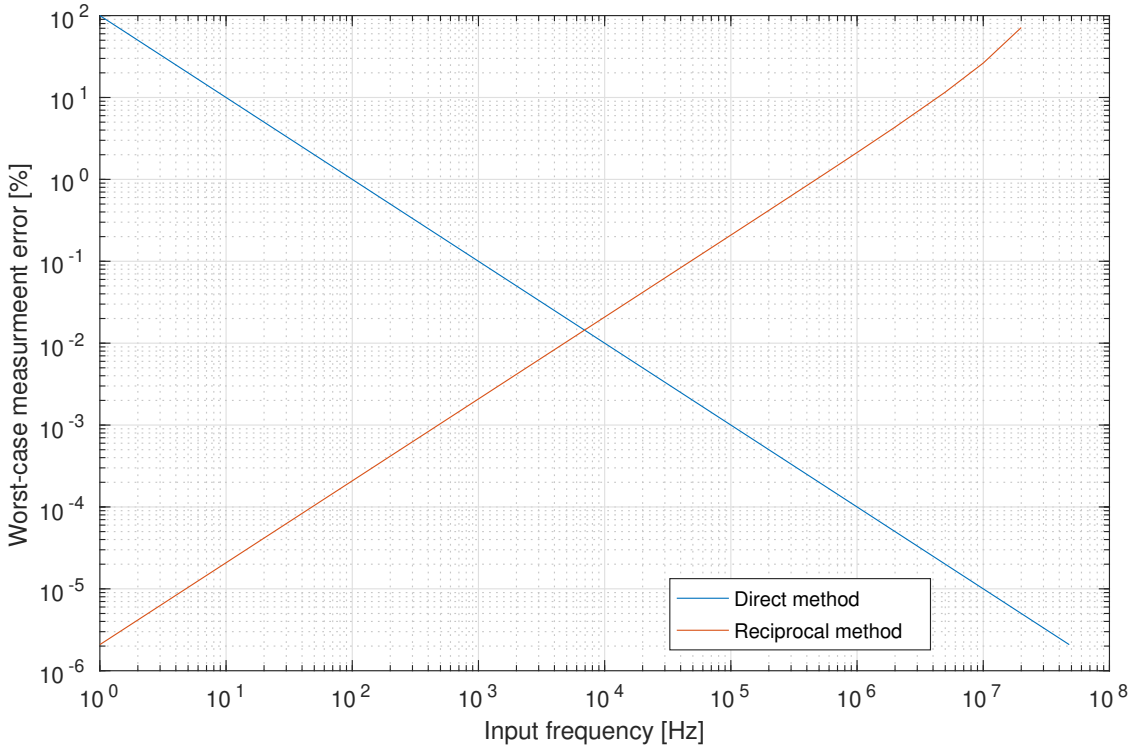


Figure 8.3: Worst-case error using the two frequency measurement methods with an ideal 48 MHz timer clock. The crossing lies at 7 kHz with an error of 0.015 %, or 1.05 Hz.

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. These can be roughly divided into DC and alternating current (AC) or time-changing signals. Analog signals are converted to digital values using ADCs. 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 mains interference (50 Hz or 60 Hz), it is 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 called the *SAR type ADC*, after its main component, the successive approximation register (SAR). A diagram of this ADC is shown in Figure 8.4.

The SAR type converter uses a DAC, controlled by the value in the SAR, which approximates the input voltage, bit by bit, following the algorithm described in [40] and outlined below:

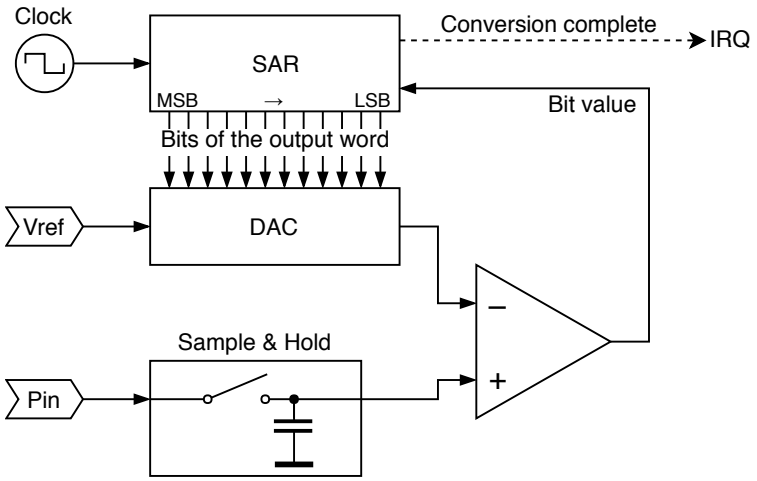


Figure 8.4: A diagram of the SAR type ADC

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. After finding all bits of the data word, an interrupt request (IRQ) is generated and the application program can read the result 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, intuitive implementation of the waveform generator is illustrated in Figure 8.5. This approach has its advantages: it is simple and works autonomously, with

no interrupt handling or interventions from the program. It could be implemented without the use of Direct Memory Access (DMA) as well, using a loop periodically updating the DAC values; of course, such approach is less flexible and we would run into problems with interrupt handling affecting the timing accuracy.

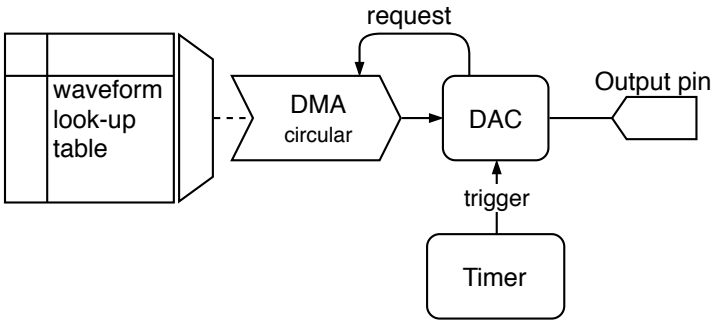


Figure 8.5: A simple implementation of the waveform generator, using DMA and a look-up table

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 predefined value, after which the counting register is reset. The counting speed is derived from the system clock frequency f_c using a prescaler 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 fine tuning, or on-line frequency and phase changes are required. Those are the strengths of Direct Digital Synthesis (DDS), an advanced digital waveform generation method well explained in [41].

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). The 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.

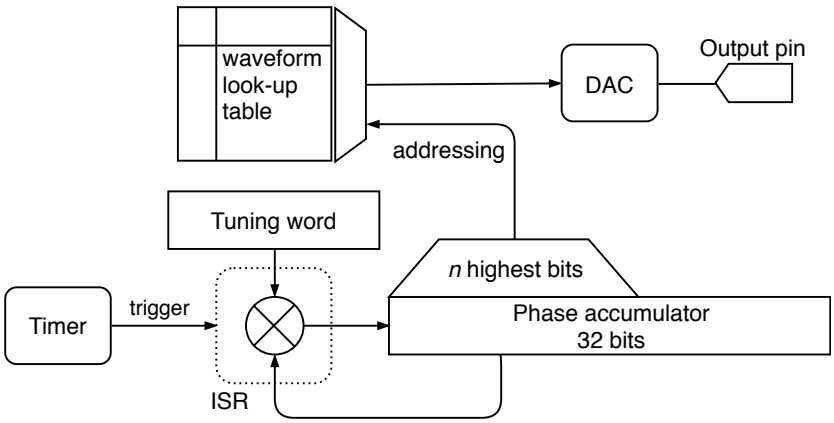


Figure 8.6: A block diagram of a DDS-based waveform generator

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 at 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., the AD9833 from Analog Devices. As the software implementation depends on a periodic interrupt, it is 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 STM32F072 microcontroller includes a Touch Sensing Controller (TSC) peripheral block. This device is meant to be used in touch-based user interfaces, e.g., for kitchen appliances or toys. We include it in GEX to serve as a demonstration of capacitive touch sensing, and it could possibly be used for simple capacitive sensors as well, such as a water level measurement.

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.

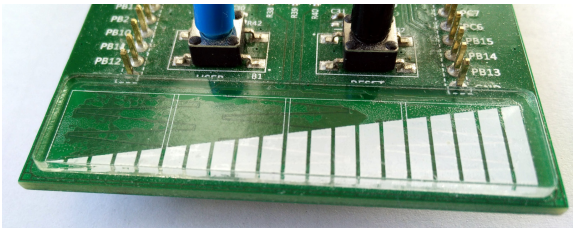


Figure 8.7: The touch slider on a STM32F072 Discovery board

The principle of capacitive touch sensing using the TSC is well explained in the microcontroller's reference manual [42], the TSC product training materials [43, 44] and application notes from ST Microelectronics [45, 46, 47, 48]. A key part of the TSC is a set of analog switches which can be combined to form several different signal paths between external pins, V_{DD}, 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.

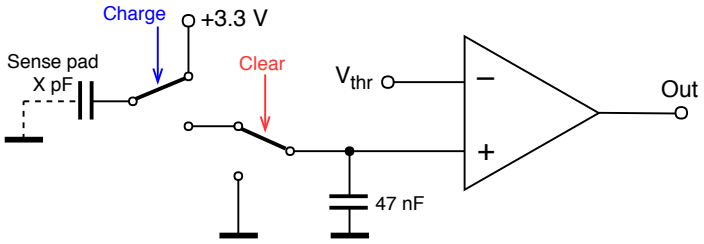


Figure 8.8: A simplified schematic of the touch sensing circuit

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

1. The sampling capacitor is discharged by connecting its free end to GND.
2. The sensing pad is connected to V_{dd} (+3.3 V) and, acting as a capacitor, charged to this voltage. 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 until 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.

A real voltage waveform measured on the sensing pad using an oscilloscope is shown in Figure 8.9.

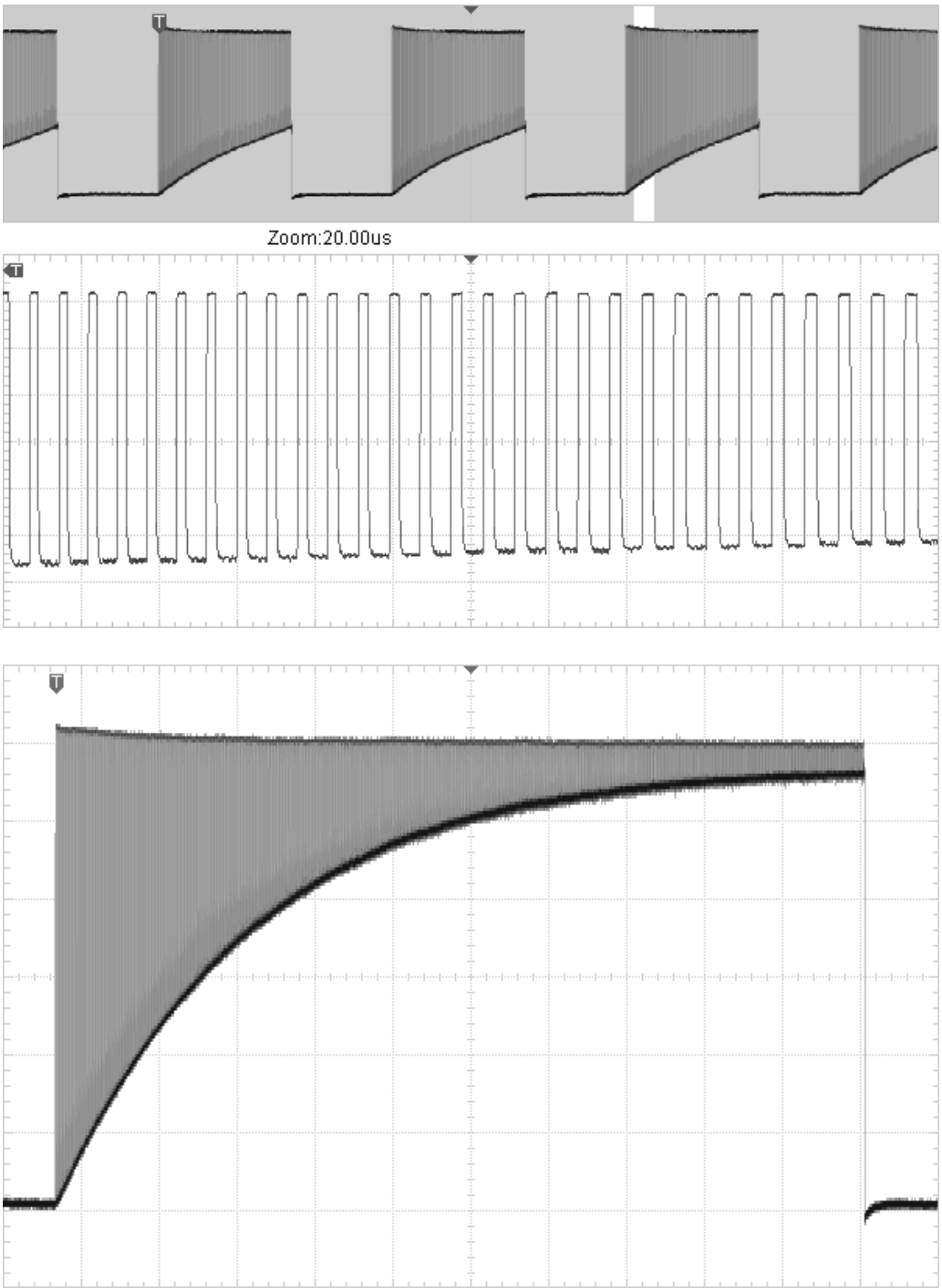


Figure 8.9: A voltage waveform measured on the touch sensing pad. The bottom side of the envelope equals the sampling capacitor's voltage—this is the phase where both capacitors are connected. The detailed view (middle) shows the individual charging cycles. The bottom screenshot captures the entire waveform, left to continue until a timeout, after the analog comparator was disabled.

Part III

Implementation

Chapter 9

Conceptual Overview

GEX is designed to be modular and easy to extend. The user-facing functionality is composed of independent software modules, called *functional blocks* or *units*, which can be configured by the user to fit their application needs. Units implement low-level logic to work with hardware peripherals of the microcontroller, and expose this functionality to the client application, running on the PC, through a communication interface. A diagram showing the entire stack, from the user application down to hardware peripherals, is shown in Figure 9.1.

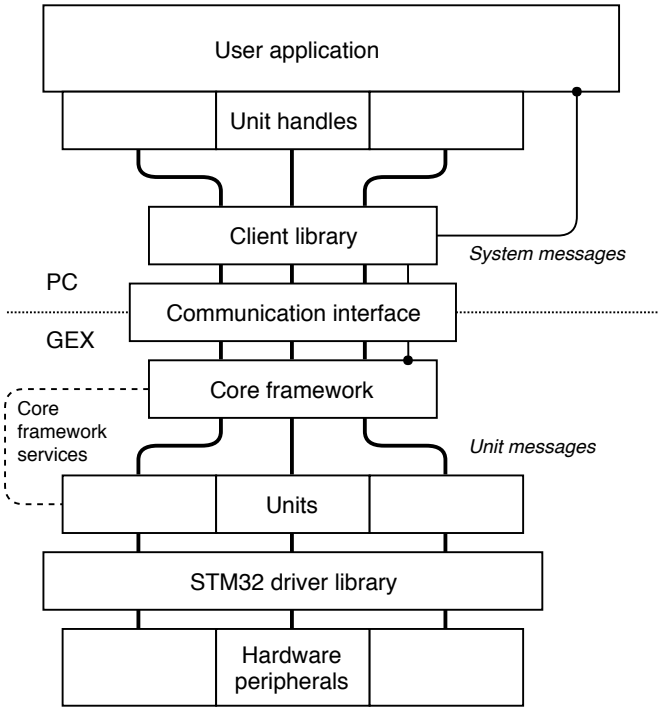


Figure 9.1: The "GEX stack", from a user application down to hardware

When we work with GEX, it is through units. The platform without units would be just an empty shell, the bare core framework; this underlying system will be described in Chapter 10. We will explore the individual units in Chapter 15, after going through the hardware realizations in Chapter 14 and covering the communication protocol in Chapter 12.

9.1 Physical User Interface

The firmware can be flashed to a STM32 development board, or a custom PCB. The particulars of those form factors will be discussed in Chapter 14.

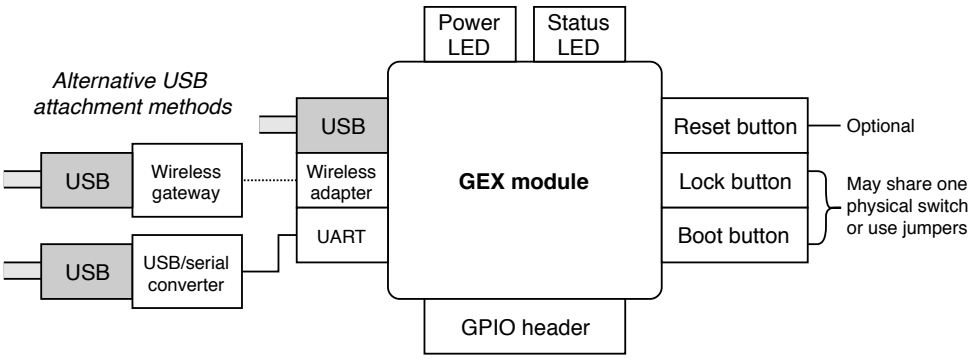


Figure 9.2: Physical user interface of a GEX module

All GEX hardware platforms have some common characteristics (Figure 9.2):

- **Power LED** – a simple indication that the board is powered on
- **Status LED** – periodic flashing every 3 s indicates correct operation, continuous light a software error[1]; other light patterns may be shown as feedback to user actions or received commands
- **Reset button** – resets the MCU; this is particularly useful during firmware development as an alternative to re-connecting the USB cable
- **Lock button** – enables or disables access to configuration files through the virtual mass storage device
- **Boot button** – when held during restart (that is, while the reset button is released), the Device Firmware Update (DFU) mode [49] is activated and a new firmware image can be flashed over the USB connection using `dfu-util` [50] or other firmware update application
- **GPIO header** – a pin header exposing the MCU's GPIO pins to be connected to external circuitry
- **Communication interface** – a connection to the host PC; multiple options may be available to choose from, a direct USB connection being the primary and always available option

9.2 GEX-PC Connection

Figure 9.2 shows three ways to connect the module to a PC. Each communication interface has its advantages and drawbacks, and is suitable for different use-cases.

[1] The microcontroller will then automatically restart within a few seconds due to a watchdog timeout.

- **Direct USB connection**

  This is the primary and most straightforward connection method. We use the CDC/ACM and MSC USB classes to have the module appear as a virtual serial port and a mass storage device, as described in Section 4.3. This method is the fastest of the three and works out-of-the-box on Linux and MacOS. On MS Windows it may require the right software driver to be installed and assigned manually[2].

- **Hardware UART**

  The hardware UART used as a communication interface is mapped to pins PA2 and PA3 to be compatible with the built-in USB/UART converter on STM32 Nucleo development boards. This interface is functionally identical to the CDC/ACM connection, but the physical UART is necessarily slower and does not implement flow control.

- **Wireless connection**

  A wireless connection is implemented using a radio module on the GEX board. To use it we need a counterpart, the *wireless gateway*, which connects to the PC via USB and acts as a CDC/ACM device.

The USB connection is always enabled first on start-up. GEX waits its for enumeration by the host PC. When not enumerated in a few seconds, it concludes that the interface is not active and tries other enabled options. The wireless module, connected through SPI, can be detected by reading one of its registers that should have a known value. A UART interface cannot be tested so reliably, thus it is always considered active[3].

9.3 Controlling GEX

GEX is a platform providing access to low-level hardware to high-level applications. However, this "high level" is relative. As was shown in Figure 9.1, the "GEX stack" ends with a *client library*, a software library used by the *user application*.

The communication protocol (one level lower in the diagram) is robust and well defined, making it possible to implement alternative client libraries in other programming languages, or for yet-unsupported platforms. This protocol is explained in Chapter 12. The client library implements the communication protocol and gives the user application access to GEX units via *unit handles*.

Any logic above the client library is in the hands of the user, which means that, to use GEX, they have to program a user application. Software libraries in languages C and Python are provided, and will be explained in Chapter 16. The Python library is easy to use even for beginner programmers, though we have to acknowledge that some users

[2]The STM32 Virtual COM port driver [51] has been tested to work with GEX on MS Windows version 7 and 8, though it must be manually assigned to the device in the Device Manager. MS Windows 10 and later should support CDC/ACM natively.

[3]A detection of the UART connection would be possible by measuring the Rx pin voltage, which should idle at a high level (here 3.3 V). This was not implemented in the initial firmware version.

might not be familiar with programming at all. Making GEX more accessible to those users, e.g., through a graphical desktop application, is an appealing idea that is certainly worth pursuing in later work.

9.4 Device Configuration

The core framework and each of the units have a number of adjustable options determining their behavior. Those settings are internally stored in a binary form, but to make their adjustment comfortable for the user, they are mapped to text configuration files in the INI format.

9.4.1 INI File Format

INI files are, in our implementation, simple text files containing three basic syntax elements: comments, sections, and key-value entries. Sections group the key-value pairs into logical blocks, e.g., the configuration of individual units.

- **Comments** start with the hash symbol (`#`) and end at the end of line
- **Sections** are textual labels enclosed in square brackets (`[UNITS]`)
- **Key-value entries** are composed of a label, the equals sign, and its value; values may be text strings, decimal or hexadecimal numbers, lists of numbers separated by commas, or any other format appropriate for the particular key

An example of the INI syntax is shown below.

```
# comment
[section]
a = 123
b = 0xFF
port = A
pins = 1,2,3
```

9.4.2 Configuration Files Structure

The configuration is split into two files: `UNITS.INI` and `SYSTEM.INI`. The system configuration file has a simple structure and does not need much explanation beyond the comments already included in it; an example of its content is captured in Listing 1. The other file, as its name suggests, serves to configure GEX units.

The units file, illustrated in Listing 2, is more complex, and *interactive*. The top part, a `[UNITS]` section, lists all available unit types. A unit is created by writing its name (an arbitrary label composed of letters, numbers, and underscore) next to the desired type. Each unit is then configured in a separate section lower in the file; however, how does one

```
## SYSTEM.INI

[SYSTEM]
# Data link accessible as virtual comport (Y, N)
expose-vcom=Y
# Show comments in INI files (Y, N)
ini-comments=Y
# Enable debug UART-Tx on PA9 (Y, N)
debug-uart=Y

# Output core clock on PA8 (Y, N)
mco-enable=N
# Output clock prediv (1,2,...,128)
mco-prediv=128

# --- Allowed fallback communication ports ---

# UART Tx:PA2, Rx:PA3
com-uart=N
com-uart-baud=115200

# nRF24L01+ radio
com-nrf=N
# Radio channel (0-125)
nrf-channel=76
# Network prefix (hex, 4 bytes)
nrf-network=12:00:09:4C
# Node address (1-255)
nrf-address=1
```

Listing 1: The `SYSTEM.INI` configuration file

know what keys are needed for which unit? This problem is solved by *interactivity* of the file.

After adding a unit name next to its type, we save the file. The disk temporarily disappears from the device list as the file's content updates. When we reopen the file, a section for the new unit will be appended for us to configure as necessary. To delete a unit, it is sufficient to remove its name from the list at the top and let the file regenerate the same way; the unit's section will disappear.

It is not uncommon that the entered (or default) configuration is invalid and the unit cannot be enabled. The error is reported by inserting a comment into the INI file, at the top of the section of the failing unit. This error message disappears when the problem is corrected.

```
## UNITS.INI

[UNITS]
# Create units by adding their names next to a type (e.g. DO=A,B),
# remove the same way. Reload to update the unit sections below.

# Digital output
DO=led
# Digital input with triggers
DI=btn1,btn2
# Neopixel RGB LED strip
NPX=
# I2C master
I2C=i2c
#...

[I2C:i2c@1]
# Peripheral number (I2Cx)
#...
```

Listing 2: Part of the `UNITS.INI` configuration file

Once we are satisfied with the configuration, it may be stored to the module's permanent memory. This is done by pushing the Lock button again, which also deactivates the virtual storage device.

It may be interesting to know that the configuration files can also be read and modified through the communication interface. A simple configuration editor (Figure 9.3) was developed to demonstrate this feature. Besides applications like this, we can use the programmatic configuration access to change GEX settings automatically by the client application. The changes may be persisted by a command, but that is not required, which lets us use them temporarily without modifying the stored configuration.

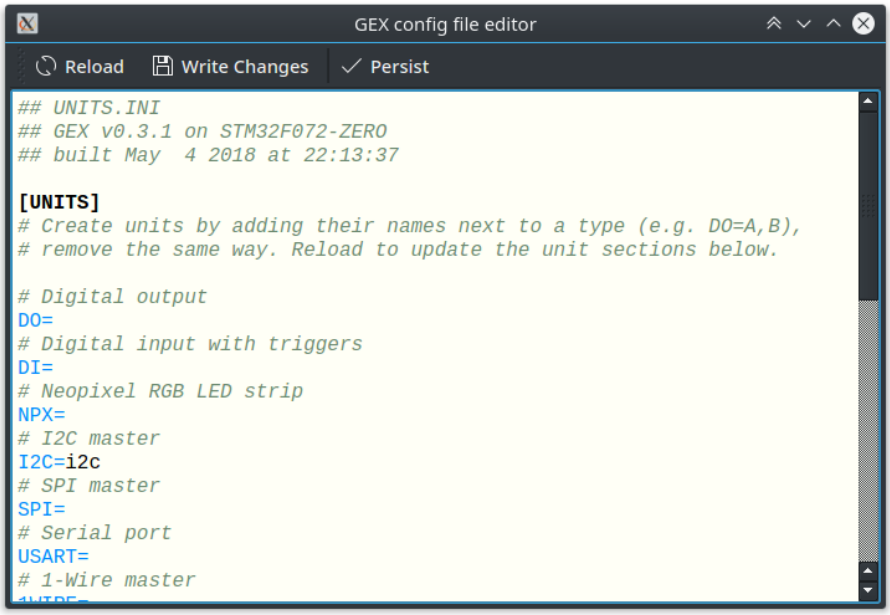


Figure 9.3: Configuration file editor GUI built using the GEX client library and PyQt4

Chapter 10

Internal Application Structure

The firmware is built around a *core framework* which provides services to units, such as the settings storage, resource allocation, message delivery, and periodic updates. In this chapter, we will focus on the structure of this framework and the services provided by it.

10.1 Internal Structure Block Diagram

The data flows and other internal logic of the firmware are depicted in Figure 10.1, with more explanation following in this chapter. The interchangeable role of the three communication interfaces can be clearly seen in the diagram, as well as the central role of the message queue, which decouples interrupts from the processing thread.

The framework provides the following services to units:

- Hardware resource allocation (Section 10.3)
- Settings storage and loading (Section 10.4)
- Unit life cycle management (Section 10.2)
- Message sending and delivery (Section 10.5)
- Interrupt routing (Section 10.6)

10.2 Unit Life Cycle and Internal Structure

GEX's user-facing functions, units, are implemented in *unit drivers*. Those are independent modules in the firmware that the user can enable and configure, in one or more instances. In practice, we are limited by hardware constraints: i.e., there may be only one ADC peripheral, or two SPI ports. The assignment of those hardware resources to units is handled by the *resource registry* (Section 10.3).

Each unit is identified by a name and a *callsign*, which is a number that serves as an address for message delivery. A unit is internally stored as a data object with the following structure:

- Name

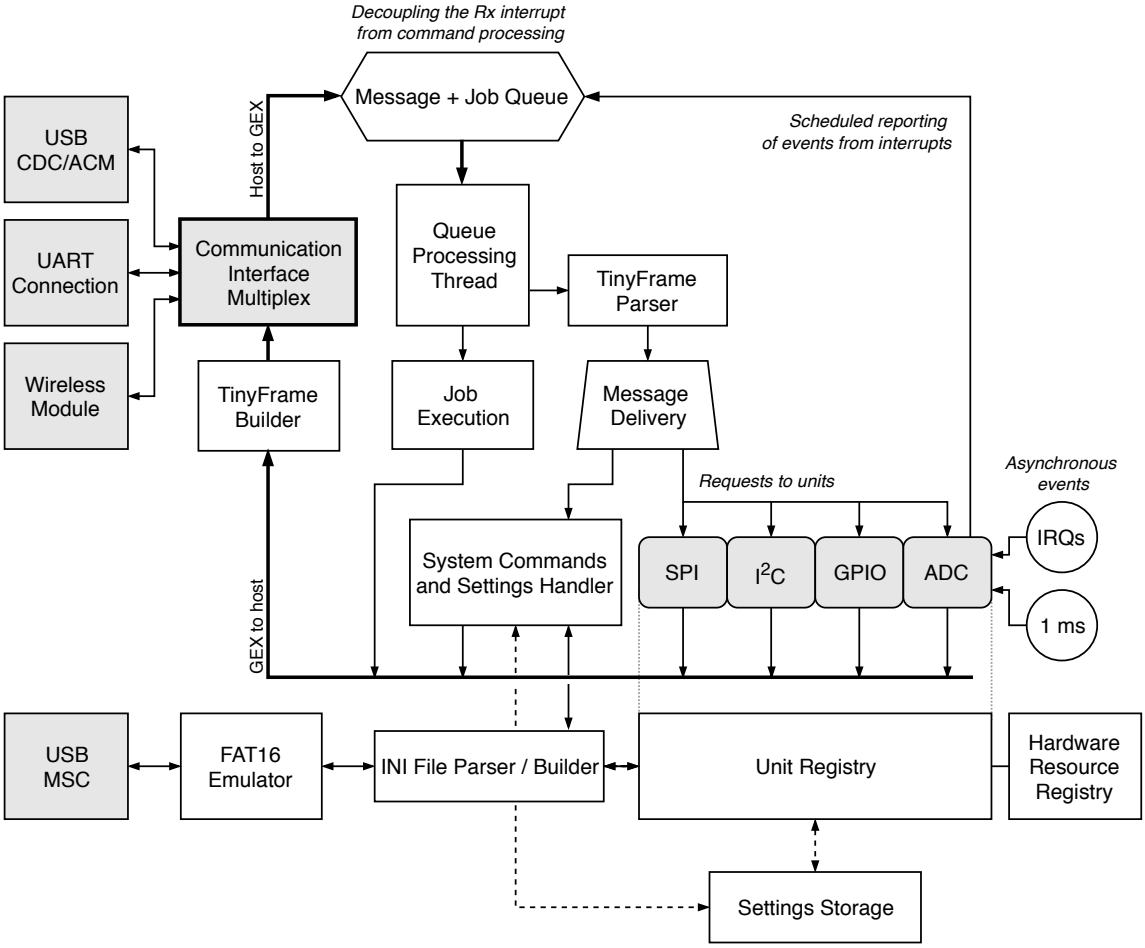


Figure 10.1: Block diagram showing the internal logic in the GEX firmware

- Callsign (one byte)
- 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 its function, such as periodic updates and interrupt handling.

When the units configuration file is modified, all units are de-initialized and removed. The binary settings are then updated based on the new values, verifying that the requested resources are available, and the units that can be enabled are subsequently initialized and made available to the user.

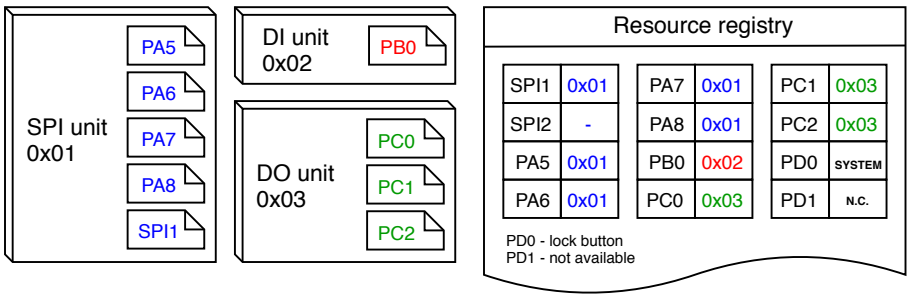


Figure 10.2: An example allocation in the resource registry

10.3 Resource Allocation

The microcontroller provides a number of hardware resources that require exclusive access: GPIO pins, peripheral blocks (SPI, I^2C, UART…), and DMA channels. If two units tried to control the same pin, the results would be unpredictable; similarly, the output of a multiply-accessed serial port could become a useless mix of the different data streams.

To prevent multiple access, the firmware includes a *resource registry* (Figure 10.2). 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 (e.g., 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.

10.4 Settings Storage

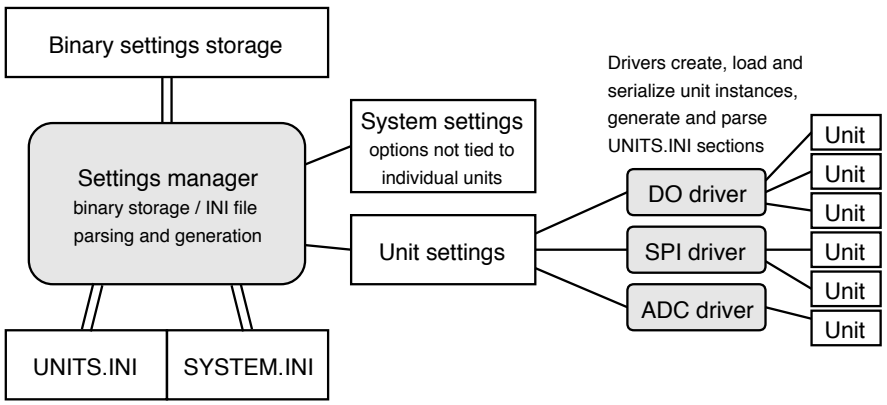


Figure 10.3: Structure of the settings subsystem

The system and unit settings are stored, in a binary form, in designated pages of the microcontroller's flash memory. The serialization and parsing of unit settings is implemented by the respective unit drivers. Figure 10.3 shows the organization of the settings subsystem; note that the "Settings manager" block has been omitted in Figure 10.1 for clarity, to better represent the data flows.

The settings persist in the flash memory even after a firmware update, which might add or remove some data fields, or otherwise rearrange the binary structure. Assuming the general layout remains unchanged, this mainly concerns the data areas holding unit (and system) settings. Backward compatibility is achieved by prefixing each storage area with its version number. When the settings are loaded by an updated firmware, it always first checks the version field and decides which format to use to parse the saved data.

10.5 Message Passing

One of the key functions of the core framework is to deliver messages from the host PC to the right units. The *TinyFrame* protocol is used, described in detail in Chapter 12; it is represented by the "TinyFrame Parser" and "TinyFrame Builder" blocks in Figure 10.1[1].

Two groups of messages exist: *system messages* and *unit messages*. System messages can, for instance, access the INI files, or request a list of available units. Unit messages are addressed to a particular unit, and their payload format is defined by the unit's driver. An incoming message is inspected and delivered to the appropriate recipient, or responded to with an error message.

In addition to message delivery, the core framework also provides event reporting. Events are messages generated by units and sent to the host to notify it about asynchronous events.

This high-level functionality resides above the framing protocol, which will be described in Chapter 12. The message format is shown in Section 12.7.

10.6 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 Arm Cortex-M0 the interrupt handlers table, defining which routine is called for which interrupt, is stored in the program memory and cannot 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.

[1] The framing library is not split into those two blocks in the source code, but the parts are functionally independent

Let us have a look at a sample interrupt handler, in this case serving 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 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 de-initializes a unit, it removes all interrupt handlers it used, freeing the redirection table slots for other use.

10.7 FreeRTOS Synchronization Objects Usage

The firmware is built around FreeRTOS (Chapter 5) and a number of its synchronization objects and patterns are used to make its operation more robust.

10.7.1 Message and Job Queue

The message and job queue, seen in Figure 10.1, is used to decouple asynchronous interrupts from message transmission. All three communication interfaces use interrupts for the asynchronous handling of incoming messages. The same interrupt handler receives an event after a transmission was completed. The queue ensures that messages can be received during the transmission of a large response that demands the use of multiple messages.

10.7.2 Lock Objects

The “transmission complete” interrupt signals the readiness to transmit more data to the message-sending task using a binary semaphore. This semaphore is taken by the task before sending a block of data, and released by the ISR, ensuring that the task waits for the transmission to complete before attempting to send more data.

Two mutexes are used in the firmware: one that guards access to the TinyFrame Builder until the previous message was fully transmitted, and one to guard a shared memory buffer (reused in several places to save memory and avoid its re-allocation). The hardware resource registry (explained in Section 10.3) does not need mutexes for individual resources, as concurrent access to those fields can never happen thanks to the way the system is organized; resources are always taken or released sequentially by the same task.

Chapter 11

Working with the GEX Source Code

Understanding the GEX source code layout is important before attempting to implement any changes or to port it to a different microcontroller type. The directory layout is shown in Figure 11.1.

The GEX core framework resides in the User folder, and units are defined in User/units. Each unit driver must be registered in the file `platform.c`. The header file `plat_compat.h` defines platform-specific constants and macros, defining parameters such as pin assignments or the clock speed. The User folder is actually a Git submodule called "gex-core" and is kept as a separate project; platform-specific customizations are managed using compile flags passed from the Makefile.

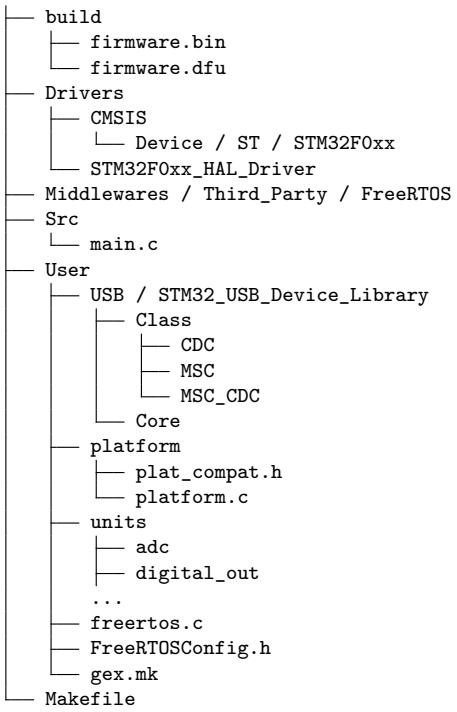


Figure 11.1: The general structure of the source code repository

11.1 Porting to a New Platform

When porting GEX to a new platform, the basis of the project can be generated by the STM32CubeMX code generator [52], using the Makefile output preset. We have to enable FreeRTOS, select a USB class (the choice does not matter, e.g., CDC/ACM can be used), and configure the system clock.

The configuration dialog gives a choice between the LL (Low Level) and HAL driver libraries; the HAL library uses a lot of program memory and often contains software bugs, while the LL library is leaner but harder to use. The LL library was used in the STM32F072 port for its smaller size.

Some files generated by STM32CubeMX were moved into the User folder (e.g., the FreeRTOS configuration and initialization, or the system time base generation), as they are mostly platform-independent. The modified USB Device library was copied here as well, as it had to be modified to support the definition of a composite class with custom descriptors.

The rest of the porting process, after generating the base project, can be summarized in the following bullet points. These are written as a checklist for the developer working on a new port; an existing, functional port may be consulted as a reference during the porting process.

- Initialize the project folder as a Git repository.
- Add the "gex-core" Git submodule as the `User` folder, and create a Git branch in it for the new platform.
- Fix the Makefile generated by STM32CubeMX; it usually contains duplicate entries in the file lists and other errors. Ensure the build ("`make`" invocation in the terminal) succeeds before making any other changes.
- Delete the USB Device library from the `Middlewares/ST` folder; GEX uses the modified version included in `User/USB`.
- Move the GPIO, FreeRTOS, USB, and other peripheral initialization from `Src` and `Inc` aside for later reference; the code in those folders should only configure the system clock, call the GEX initialization function "`GEX_PreInit()`", and start FreeRTOS with "`MX_FREERTOS_Init()`" and "`osKernelStart()`".
- Add "`include User/gex.mk`" and "`GEX_PLAT=MYPLATFORM`" at the top the Makefile, with the desired platform name in place of "`MYPLATFORM`"; the name must be a valid C identifier.
- Add "`GEX_CFLAGS`", "`GEX_SOURCES`", "`GEX_INCLUDES`", "`GEX_CDEFS`" into the appropriate file lists in the Makefile; those variables are exported from `User/gex.m` and contain lists of GEX source files and compiler flags.

  Use "`$(foreach x,$(GEX_SRC_DIR),$(wildcard $(x)/*.c))`" to include all source files from "`GEX_SRC_DIR`" in the "`C_SOURCES`" list.
- Remove all definitions of "`Error_Handler()`", "`FULL_ASSERT`" and "`assert_param()`" from the files left inside `Inc` and `Src`, and add "`#include "stm32_assert.h"`" to `Src/main.h`. GEX uses the functions declared in `User/stm32_assert.h` for assertions.
- Update `User/FreeRTOSConfig.h` and `User/platform/plat_compat.h` for the new platform. Preprocessor directives like "`#ifdef`" are used to define configuration applicable only to one platform, without affecting others.
- Update `User/platform/platform.c` to register the initially supported units; a good choice is the DO (Digital Output) unit that is straightforward to update and can be used to verify the platform's functionality.

  Define a flag like "`UNIT_DO=1`" at the top of the Makefile for each registered unit. `User/gex.m` may be used as a reference for the expected variable names. This is used to conditionally enable or disable the inclusion of the particular unit's source files in the compilation.

- Update the USB functions in the `User/USB` folder according to the ones previously generated by STM32CubeMX. The USB Device library uses these as an interface to the different USB peripheral versions.
- Update other platform-dependent code (such as the debug UART configuration). The compiler should warn about those occurrences when trying to build the project.
- Try to build the project by running "`make`" in the terminal.

After the firmware successfully compiles, we can flash it to the MCU using ST-Link by running "`make flash`", or through the DFU interface with "`make dfu`".

The first thing to verify after flashing the firmware is that the debug UART's configuration is correct and we can see the log output, which should be available on pin PA9 at 115200 baud. Thanks to a generous usage of assertions, most errors should produce helpful log messages in the log. We can then proceed to experiment with the device, testing the features described in Chapter 9 while observing the debug log for any anomalies.

Chapter 12

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 use the same binary messaging protocol and are functionally interchangeable.

GEX uses the *TinyFrame* [53] framing library, 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, including checksum calculation, and provides high-level API.

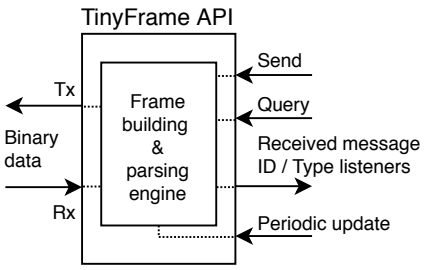


Figure 12.1: TinyFrame API

Both peers, GEX and the client library running on the host PC, are at an equal level: either side can independently send a message at any time. The communication is organized in transactions; a transaction consists of one or more messages going in either direction. A message can be stand-alone, or chained to another, typically a request, using the frame ID field; this is the major advantage over text-based protocols, like AT commands, where all messages are independent and their relation to each other is not always clear.

12.1 Binary Payload Structure Notation

Binary payloads are described in several places of this text. We use a shortened notation derived from the C language to represent field data types:

- `bool` – 8-bit field allowing values 0 (false) and 1 (true)
- `u8`, `u16`, `u32` – unsigned 8-, 16-, or 32-bit integer
- `i8`, `i16`, `i32` – signed (two's complement) 8-, 16-, or 32-bit integer
- `char` – an 8-bit ASCII character
- `float` – single-precision (32-bit) IEEE 754 [54] floating point number
- `double` – double-precision (64-bit) IEEE 754 [54] floating point number
- `u8[]` – array of variable length
- `u8[n]` – array of length n
- `cstring` – zero-terminated character string (like `char[]`, ending with a 0x00 byte)

12.2 Frame Structure

Message frames have the following structure (all little-endian):

"TinyFrame" frame structure, as used in GEX

- `0x01` start-of-frame marker
- `u16` frame ID
- `u16` payload length
- `u8` frame type
- `u8` header checksum
- `u8[]` payload
- `u8` payload checksum (omitted for empty payloads)

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.

12.3 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 in TinyFrame; its values are defined by the application. 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, but has a lower priority.

Each message can be handled by only one listener, unless the listener explicitly requests it to be passed on. Messages not handled by any listener are given to a default listener, which can, e.g., write an error to a debug log.

12.4 Designated Frame Types

Table 12.1 lists the frame types defined by GEX. It is divided into four logical sections: General, Bulk Read/Write, Unit Access, and Settings. The payloads belonging to those frame types will be outlined in the following sections.

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 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*

Table 12.1: Frame types used by GEX

12.5 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 would not be called for the subsequent messages.

Figure 12.2 shows a diagram of the bulk read and write data flow.

12.5.1 Bulk Read

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

Frame 0x21 (INI Read) payload structure
• `u8` which file to write – 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):

Frame 0x03 (Bulk Read Offer) payload structure
• `u32` full size of the file in bytes • `u32` 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:

Frame 0x04 (Bulk Read Poll) payload structure
• `u32` how many bytes to read (at most)

Frame 0x06 (Bulk Data) or 0x07 (Bulk End) payload in a "read" transaction
• `char[]` a chunk of the read file

12.5.2 Bulk Write

To overwrite an INI file, we first send a frame 0x22 (INI Write) with the file size as its payload. The name of the file is irrelevant, as it is detected automatically by inspecting the content.

Frame 0x22 (INI Write) payload structure
• `u32` size of the written file, in bytes

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

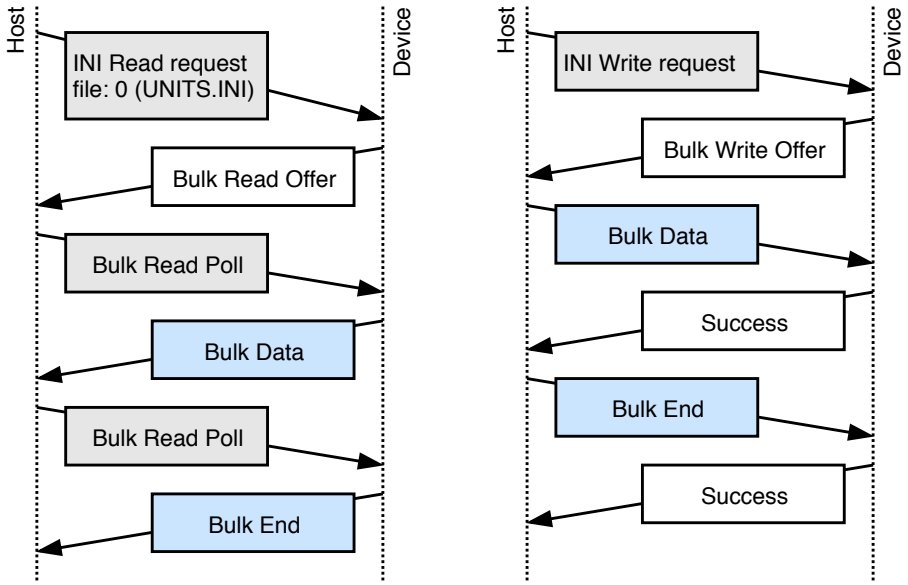


Figure 12.2: A diagram of the bulk read and write transaction.

Frame 0x05 (Bulk Write Offer) payload structure
• `u32` total bytes to write (here copied from the request frame) • `u32` how many bytes may be written per message

We can now send the file as a series of frames of type 0x06 (Bulk Data), or 0x07 (Bulk End) in the last frame, with chunks of the data as their payloads. Each frame is confirmed by 0x00 (Success).

Frame 0x06 (Bulk Data) or 0x07 (Bulk End) payload in a "write" transaction
• `char[]` a chunk of the written file

12.5.3 Persisting the Changed Configuration to Flash

The written INI file is immediately parsed and the settings are applied. However, these 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 closing the virtual mass storage by pushing the Lock button.

12.6 Reading the 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:

Frame 0x20 (List Units) response structure

- `u8` the number of available units
- For each unit:
  - `u8` unit callsign
  - `cstring` unit name
  - `cstring` unit type

12.7 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.

12.7.1 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:

Frame 0x10 (Unit Request) payload structure

- `u8` unit callsign
- `u8` command number, handled by the unit driver
- `u8[]` command payload, handled by the unit driver; its size and content depend on the unit driver and the particular command number, as defined in Chapter 15

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 flag, as that would produce two responses at once.

12.7.2 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):

Frame 0x11 (Unit Report) payload structure

- `u8` unit callsign
- `u8` report type, defined by the unit driver
- `u64` event time (microseconds since power-on)
- `u8[]` report payload; similar to requests, the payload structure depends on the unit driver and the particular report type, as defined in Chapter 15

Chapter 13

Wireless Interface

Four methods of a wireless connection have been considered: Bluetooth (perhaps with the Texas Instruments CC2541), WiFi with the Espressif ESP8266, a 868 MHz long range radio link with the Semtech SX1276, and a 2.4 GHz radio link with the nRF24L01+. Bluetooth was dismissed early for its complexity, and the ESP8266 for its high power consumption, although both solutions might be viable for certain applications and with more development time.

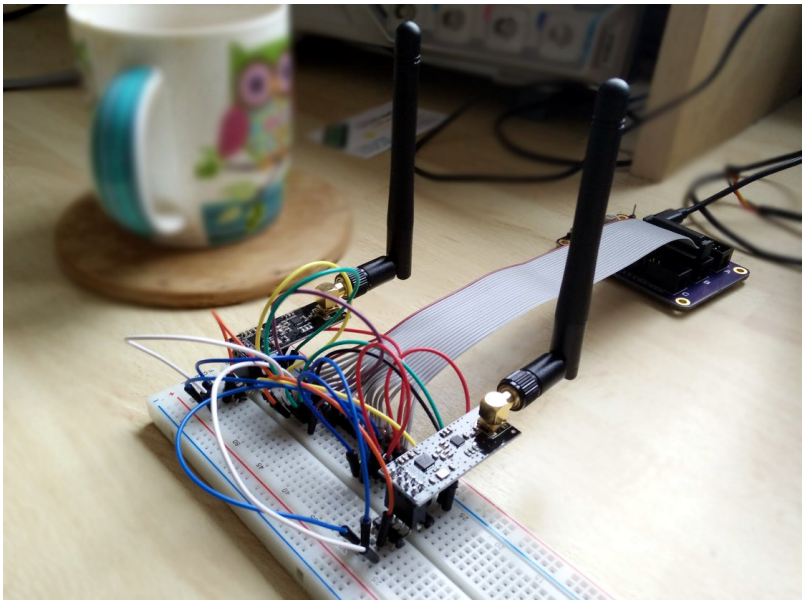


Figure 13.1: Test setup with a GEX prototype controlling two nRF24L01+ modules

13.1 Modulations Overview

A brief overview of the different signal modulation techniques is presented here to aid the reader with understanding of Table 13.1 and the rest of the chapter.

13.1.1 On-Off Keying (OOK)

In on-off keying (OOK), the carrier generator is switched on and off to transmit ones and zeros.

13.1.2 Frequency Shift Keying (FSK)

Frequency-shift keying (FSK) uses a change of the carrier frequency to transmit data. The simplest form of FSK is binary frequency-shift keying (BFSK), which uses a pair of alternating frequencies to transmit ones and zeros.

13.1.3 Gaussian Frequency Shift Keying (GFSK)

Gaussian frequency-shift keying (GFSK) is an improvement over basic FSK which does not switch between the different frequencies instantaneously, but uses a Gaussian filter to make the changes less abrupt, which reduces the side-band interference otherwise generated by the sharp edges. This scheme can be imagined as sending the binary waveform through a Gaussian filter and then modulating a VCO with its output, rather than changing the VCO's control voltage discretely. GFSK is used in the Bluetooth standard.

13.1.4 Minimum-Shift Keying (MSK)

Minimum-shift keying (MSK) is another FSK-based modulation scheme. In MSK, the frequencies representing different symbols are chosen such that there are no sharp changes in the phase of the output waveform, the modulation is *phase-coherent*. This is another way to reduce side-band interference.

13.1.5 Gaussian Minimum-Shift Keying (GMSK)

Gaussian minimum-shift keying (GMSK) is a variant of MSK which uses a Gaussian filter to shape the digital signal before sending it to the oscillator. The principle is similar to GFSK, and it is a yet another way to reduce side-band interference and increase spectral efficiency. GMSK is used in the Global System for Mobile communications (GSM).

13.1.6 LoRa Modulation

LoRa uses a direct sequence frequency hopping spread spectrum modulation scheme and can achieve very long range transmission. A higher-level specification defines how the devices using LoRa should behave in a large area network, though the modulation may be used for point-to-point connections as well.

LoRa is a proprietary technology developed by Semtech; it is free to use, but is available only with transceiver ICs produced by the company.

13.2 Comparing SX1276 and nRF24L01+

The SX1276and nRF24L01+ transceivers have both been tested using the first GEX prototype, proving its usefulness as a hardware development tool, they proved capable of fulfilling the requirements of the GEX radio link use-case. We compared them in Table 13.1

using data from their datasheets [55, 56]. It is apparent, after inspecting the table, that each has its strengths and weaknesses and the choice depends on the particular application's needs.

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.)	over 10 km	up to 1 km
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
Software reset	NRESET pin	not available
Extra	LoRa FHSS, packet engine	ShockBurst protocol engine
Price	$7.3	$1.6

Table 13.1: Comparison of the SX1276 and nRF24L01+ wireless transceivers, using data from their datasheets (price in USD from DigiKey in a 10 pcs. quantity, recorded on May 6th 2018)

The SX1276 supports additional modulation modes, including the proprietary LoRa scheme that can be received at long distances. The power consumption required to achieve this long-range communication makes it impractical for continuous operation on battery or solar power.

The 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 worked around using an external transistor to momentarily disconnect it from the power supply.

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 potentially more robust communication without additional overhead in the control software.

13.3 Wireless Link with the nRF24L01+

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, should the need for a long range communication arise.

A pair of these radio modules can form a bidirectional data connection, functionally replacing USB or UART as a communication interface. However, we need to connect the second module to the PC to control GEX through the radio link. A separate USB device, a *wireless gateway*, was developed for this purpose; its hardware will be presented in Section 14.4.

13.3.1 The Wireless Gateway

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. However, the standard GEX communication protocol cannot be used directly, as the gateway itself needs to be managed through the interface, and it can connect to more than one GEX module at once, necessitating an addressing scheme. This problem could be solved by adding a side channel, additional USB endpoints, to interact with the gateway itself; that option was not explored further, but it is clear that it would compromise our ability to use the simple virtual COM port USB driver.

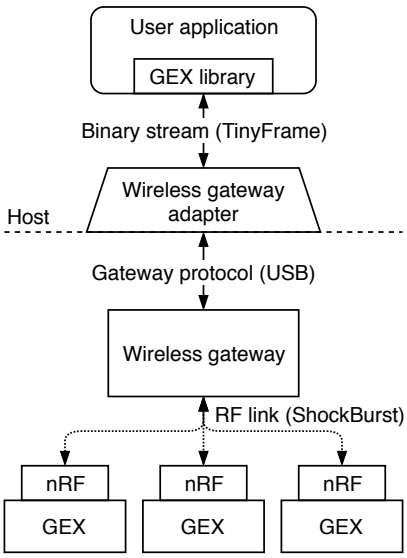


Figure 13.2: A block diagram of the wireless connection

The gateway has a 4-byte *network ID*, a number derived from the MCU's unique ID number. The network ID must be entered into the system settings file of all nodes that wish to communicate with the gateway. Additionally, each module is assigned a 1-byte number serving as its address in the network. These addresses are loaded into the gateway by the user application, and are used to configure the nRF24L01+ to listen to messages from the corresponding nodes. The nRF24L01+ uses 5-byte addressing; the full node addresses are composed of the network ID and the one additional address byte.

13.3.2 The Gateway Protocol

The gateway protocol, when used to communicate with a node, encapsulates the raw binary data sent to or from connected nodes; the wrapped TinyFrame protocol remains unchanged.

All messages sent to or from the gateway are a multiple of 64 bytes long, padded with zeros if shorter. A message starts with a control byte determining its type, as summarized in the following table listing the structure of all supported messages.

First byte	Function	Structure
'r' (114)	**RESTART** Restart the gateway, disconnecting all nodes. This is functionally equivalent to re-plugging it to the USB port.	
'i' (105)	**GET_NET_ID** Read the unique 4-byte network ID. This command has no side effects and may be used as "ping" to verify the USB connection.	*Response:* • `0x01` • `u8[4]` network ID
'n' (110)	**ADD_NODES** Configure the gateway to listen for messages from the given nodes. Nodes may be removed using the RESTART command.	*Request:* • `u8` count • `u8[]` node addresses
'm' (109)	**SEND_MSG** Send a binary message to one of the connected nodes. The message may span multiple 64-byte frames; the subsequent frames will contain only the payload bytes, or zero padding at the end of the last one.	*Request:* • `u8` node address • `u16` length • `u8` checksum (inverted XOR of all payload bytes) • `u8[]` payload
0x02	**INCOMING_MSG** A message was received from one of the configured nodes. This is an event frame sent by the gateway to the host.	*Payload:* • `u8` node address • `u8` message length • `u8[]` payload

13.3.3 Gateway Initialization Procedure

A host program connecting to a node or nodes through the gateway should follow the following procedure to initialize and configure the gateway:

1. Send the "GET_NET_ID" command to test if the gateway is connected (and obtain its network ID as a side effect)
2. Restart the gateway using the "RESTART" command to clean any possible previous configuration; this is not needed when the gateway was restarted or freshly connected to the USB port.
3. Add the node address(es) using the "ADD_NODES" command.
4. Ping the connected node(s) through the GEX communication protocol to test the connection. Note that GEX will not use the radio module if USB is connected and enumerated, as it has lower priority.

Chapter 14

Hardware Realization

14.1 Using a Discovery Board

It has been proposed earlier in the text that STM32 Nucleo and Discovery development boards might be used as the hardware platform for this project. Indeed, a Discovery board with STM32F072 [57] was used as a development platform for the majority of the GEX firmware. This inexpensive board may be used to try the GEX firmware without having access to the custom hardware.

14.1.1 Discovery STM32F072 Configuration and Pin Mapping

This Discovery board is fitted with four LEDs on GPIO pins PC6 through PC9, in a compass arrangement. The "north" LED, PC6, is used as the GEX status indicator. The "User" button, connected to PA0, is mapped as the GEX Lock button, controlling the settings storage.

We advise the reader, as a potential user of this discovery board, to review its schematic diagram and ensure the solder-jumpers are configured correctly:

- Jumpers SB20 and SB23 must be closed to enable the User USB connector
- Jumper SB17 must be open and SB19 closed to use the 8 MHz clock signal provided by the on-board ST-Link programmer; the internal USB-synchronized 48 MHz oscillator will be used if the clock signal is not provided (SB19 open).
- Jumpers SB27 through SB32 should be closed to connect the GPIO pins normally dedicated to the touch sensing strip to the board's header.
- Capacitors C26 through C28 are sampling capacitors for the TSC. There are, unfortunately, no jumpers available to disconnect them, and they interfere in high-speed signals on the used pins (PA3, PA7, PB1). The only solution is to carefully remove them from the board if the TSC is not needed.

An accelerometer IC L3GD20 is fitted on the board. The chip is attached to SPI2 on pins PB13 (SCK), PB14 (MISO) and PB15 (MOSI), with NSS on pin PC0, and PC1 and PC2 used for interrupt flags. This chip cannot be disconnected or disabled and it is difficult to remove; care must be taken to avoid its interference on the used pins.

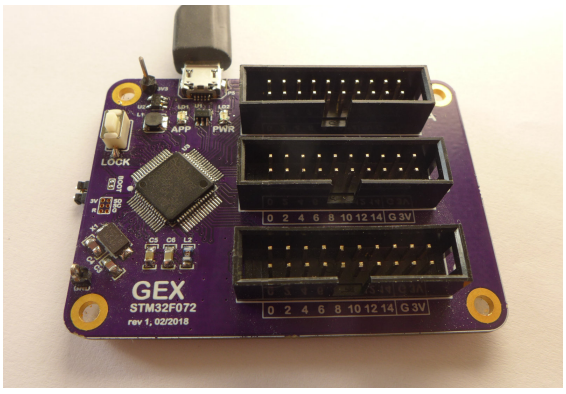
14.2 GEX Hub

GEX Hub was the first custom PCB designed for GEX. It uses the same microcontroller as the Discovery board, thus the firmware modifications needed to make it work with this new platform were minimal.

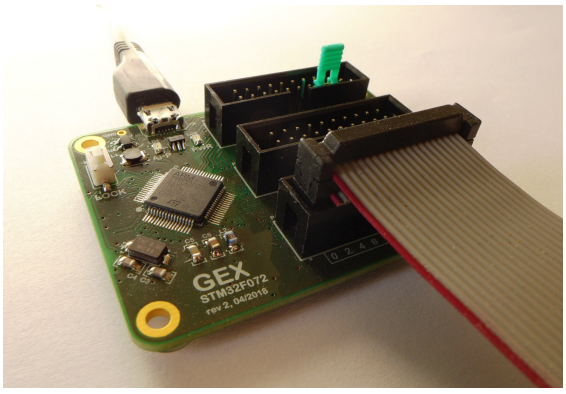
The Hub board provides access to all the GPIO pins using three flat-cable connectors, one for each port; they also contain a ground and power supply connection to make the connection of external boards or a breadboard easier, needing just one cable. The use of flat cables, however, is not mandatory—those connectors are based on the standard 2.54 mm pitch pin headers, allowing the user to connect to them using widely available "jumper wires".

This board was produced in two revisions. The original model (Figure 14.1a) proved fully functional, except for the two connectors on the left side, the boot jumper and a programming header, which had the wrong footprints and could not be populated; this mistake was fixed by soldering the jumper from the bottom of the PCB, and the programming header was never needed thanks to the USB bootloader working without issues.

The updated revision removes the two problematic footprints altogether; a reorganization in the GPIO connectors allowed them to be moved together with the other pins. Revision 1 used a dedicated header for the Boot jumper that was meant to be closed during normal operation, and removed only to enter the bootloader. Revision 2 moved the boot pin into the connector, and such arrangement would not be practical; the solution was to invert the jumper's logic by changing the Boot pull-up to a pull-down. The bootloader is now activated by inserting a regular 2.54 mm jumper into the connector[1], as can be seen in Figure 14.1b.



(a) : Revision 1



(b) : Revision 2

Figure 14.1: Two revisions of the GEX Hub module, rev. 2 shown with the boot jumper and a flat cable.

[1] A restart is required in all cases for the boot jumper changes to have effect

14.3 GEX Zero

Our desire to re-use the form factor of the Raspberry Pi Zero to exploit the existing market with add-on boards and cases for it has been revealed already in Section 2.5. This was brought to fruition with GEX Zero, the second realized prototype board (counting the two revisions of GEX Hub as one).

GEX Zero exactly copies the dimensions of the Pi Zero, which introduces several challenges:

- It must be a one-sided board, with no components on the bottom; this is needed for acrylic cases which sit flatly against the PCB, with a cut-out for the pin header.
- Buttons and the USB connector have to exactly align with connectors on the Pi Zero to fit the openings in its cases.
- The board size is fixed, and rather small; we used only two layers to save production cost, but this proved a significant challenge and the electrical characteristics of some connections may not be ideal.
- To make use of the Raspberry Pi add-on boards, called HATs or pHATs, a particular organization of the pin header is required. This is discussed in more detail below.

14.3.1 Finding the Best Pin Assignment

Like our STM32 microcontroller, the Broadcom processor on the Raspberry Pi multiplexes its GPIO pins with alternate functions, and, likewise, each function is available only on a small selection of pins. A number of compromises had to be made to achieve maximum compatibility.

show the pi header mappings and the gex zero pin mappings

14.4 Wireless Gateway

Figure 14.4

TODO about the gateway ..

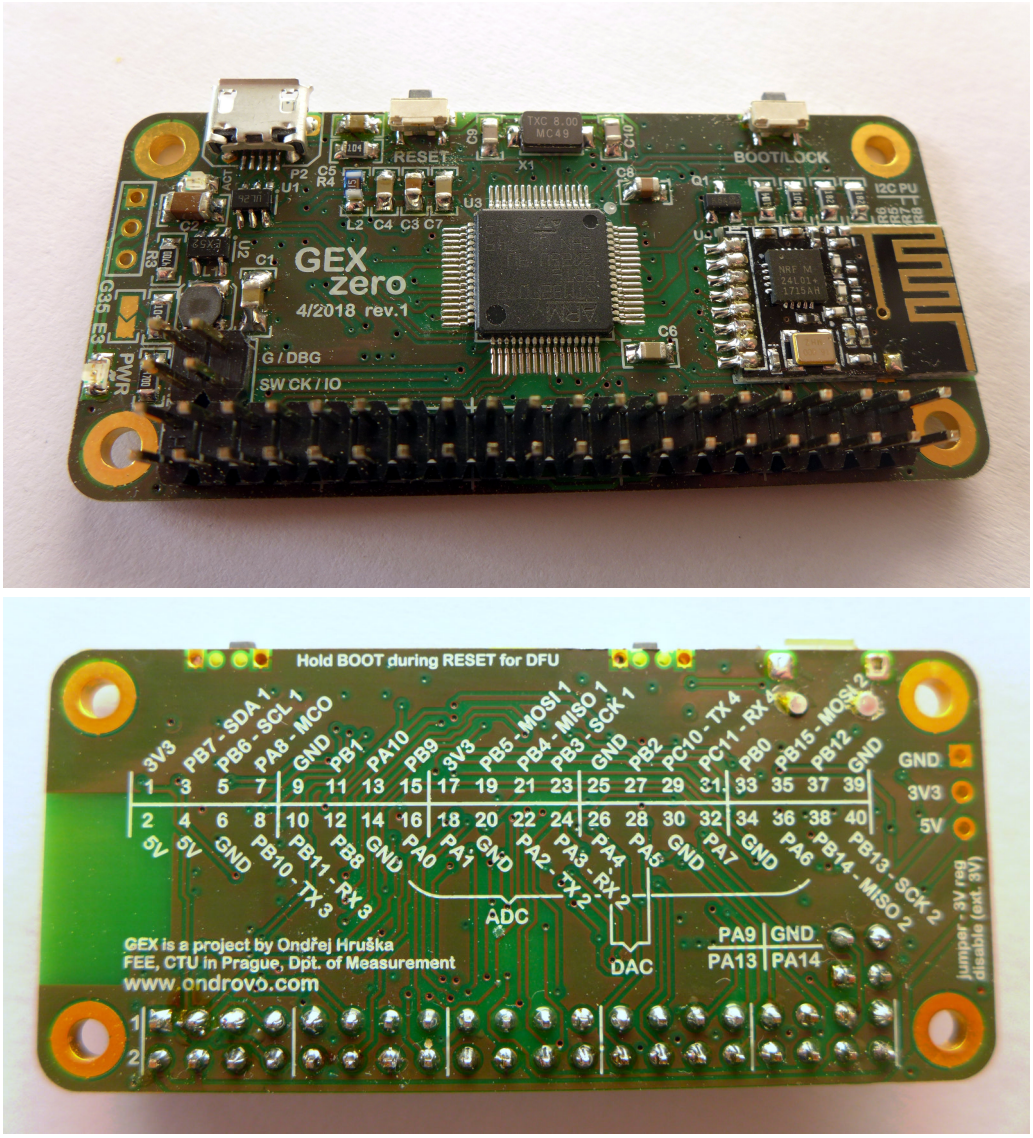


Figure 14.2: GEX Zero, top and bottom side

Figure 14.3: GEX Zero in the official Raspberry Pi Zero case and an aftermarket acrylic case

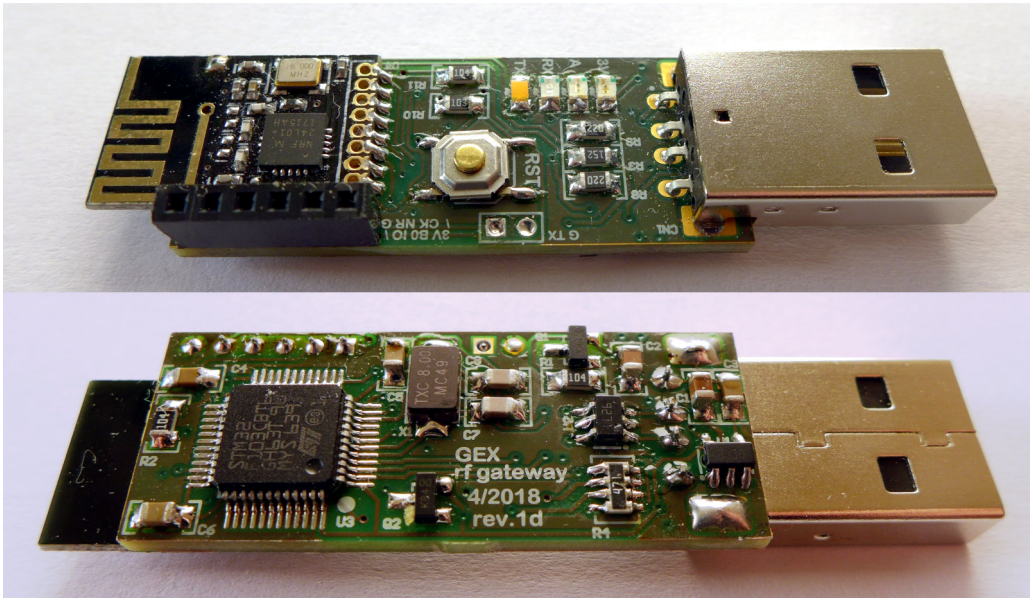


Figure 14.4: The wireless gateway module (top and bottom side)

Chapter 15

Units Overview, Commands and Events Description

This chapter describes all functional blocks (units) implemented in GEX, version 1.0. The term "unit" will be used here 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. The commands and events described here form the payload of TinyFrame messages 0x10 (Unit Request) and 0x11 (Unit Report), as described in Section 12.7.

The number in the first column of the command (or event) tables, marked as "Code", is the command number (or report type) used in the payload to identify how the message data should be treated. When the request or response payload is empty, it is 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 after the command is finished.

15.1 General Notes

15.1.1 Unit Naming

Unit types are named in uppercase (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".

15.1.2 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 is common to be left with a discontiguous group of pins after assigning all the alternate functions needed by an application.

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 15.1. The

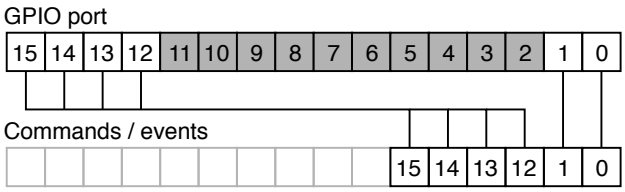


Figure 15.1: Pin packing

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.

15.2 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 timing derived from the system timebase, in order to support pulses on all pins regardless of hardware PWM support. Pins in commands are expressed in the packed format (Section 15.1.2).

15.2.1 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=
```

15.2.2 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

Code	Function	Structure
2	**CLEAR** Set selected pins to 0	*Request:* • `u16` pins to clear
3	**TOGGLE** Toggle selected pins	*Request:* • `u16` pins to toggle
4	**PULSE** Generate a pulse on the selected pins. The microsecond scale may be used only for 0–999 μs.	*Request:* • `u16` pins to pulse • `bool` active level • `u8` scale: 0-ms, 1-μs • `u16` duration

15.3 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 report asynchronous events on a pin change.

All pins of the unit may be configured either for a rising, falling, or any change detection; due to a hardware limitation, the same pin number may not be used for event detection on different ports (e.g., A1 and B1) at the same time. In order to receive a pin change event, we must arm the pin first, using a command; it can be armed for a single event, or it may be re-armed automatically with a hold-off time. It is, further, possible to automatically arm selected pins on start-up, removing the need to arm them, e.g., after the module restarts or is re-connected.

15.3.1 Digital Input Configuration

```
[DI:in@2]
# Port name
port=A
# Pins (comma separated, supports ranges)
pins=10-8,3-0
# Pins with pull-up
pull-up=10,9
# Pins with pull-down
pull-down=0

# Trigger pins activated by rising/falling edge
trig-rise=10
trig-fall=
# Trigger pins auto-armed by default
auto-trigger=10
```

```
# Triggers hold-off time (ms)
hold-off=100
```

15.3.2 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

15.3.3 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

15.4 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 74xx4094 or 74xx595. Those are commonly used to control segmented LED displays, LED user interfaces, etc.

Those devices may be daisy-chained: the output of one is connected to the input of another, sharing the same clock and other signals, and they work together as one longer shift register.

A SIPO shift register has the following pins (possibly named differently with chips from different vendors):

- *Shift* – SCK; shifts the data in the register by one bit

- *Data In* – MOSI; serial data to load into the register
- *Data Out* – output for daisy-chaining with other shift registers
- *Store* – latches the current register data and shows it on the output
- *Clear* – erases the latched data and clears the display

This unit automatically handles both the *Shift* and *Store* signals, provides access to the *Clear* output, and is capable of loading multiple shift registers in parallel (an arrangement sometimes used instead of daisy-chaining). The polarity (active level) of all signals can be configured.

It is, additionally, possible to set the data lines to arbitrary "idle" level(s) before sending the *Store* pulse; this may be latched and used for some additional feature on the user interface, such as a brightness control.

15.4.1 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
```

15.4.2 SIPO Commands

The WRITE and CLEAR_DIRECT commands are the only ones normally used. The others provide manual control over all the output signals for debugging or testing.

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 (packed pins) • For each output (same size) – `u8[]` data to load

Code	Function	Structure
1	**DIRECT_DATA** Directly write to the data pin(s)	*Request:* • `u16` values to write (packed pins)
2	**DIRECT_CLEAR** Pulse the *Clear* pin	
3	**DIRECT_SHIFT** Pulse the *Shift* pin	
4	**DIRECT_STORE** Pulse the *Store* pin	

15.5 NeoPixel Unit

The NeoPixel unit implements the protocol needed to control a digital LED strip with WS2812, WS2811, or compatible LED driver chips. The NeoPixel protocol (explained in Section 7.5) 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 (3×8 bits color), or as the little- or big-endian encoding of colors in a 32-bit format: 0x00RRGGBB or 0x00BBGGRR. The 32-bit format is convenient when the colors are already represented as an array of 32-bit integers, e.g., extracted from a screen capture or an image.

15.5.1 NeoPixel Configuration

```
[NPX:neo@3]
# Data pin
pin=A0
# Number of pixels
pixels=32
```

15.5.2 NeoPixel Commands

Code	Function	Structure
0	**CLEAR** Switch all LEDs off (sets them to black)	

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

15.6 SPI Unit

The SPI unit provides access to one of the microcontroller's SPI peripherals. The unit can be configured to any of the hardware-supported speeds, clock polarity, and clock phase settings. Explanation of those options, including diagrams, can be found in Section 7.2.

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 to prevent a short circuit between slaves transmitting the opposite logical level.

The QUERY command of this unit, illustrated by Figure 15.2, is flexible enough to support all types of SPI transactions: read-only, write-only, and read-write, with different request and response lengths and paddings. The slave select signal is asserted during the entire transaction.

15.6.1 SPI Configuration

```
[SPI:spi@5]
# Peripheral number (SPIx)
device=1
```

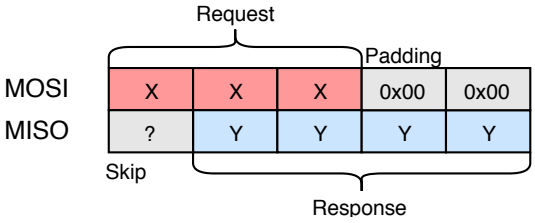


Figure 15.2: SPI transaction using the QUERY command

```
# 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
```

15.6.2 SPI Commands

Code	Function	Structure
0	**QUERY** Exchange bytes with a slave device; see the diagram in Figure 15.2	*Request:* • `u8` slave number 0–16 • `u16` response padding • `u16` response length • `u8[]` bytes to write *Response:* • `u8[]` received bytes
1	**MULTICAST** Send a message to multiple slaves at once. The "addressed slaves" word uses the packed pins format (Section 15.1.2).	*Request:* • `u16` addressed slaves • `u8[]` bytes to write

15.7 I²C Unit

The I²C unit provides access to one of the microcontroller's I²C peripherals. More on the I²C bus can be found in Section 7.3.

The unit can be configured to use either of the three standard 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; the 7 or 10 least significant bits will be used as the actual address.

15.7.1 I²C Configuration

```
[I2C:i2c@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
```

15.7.2 I²C Commands

Code	Function	Structure
0	**WRITE** Perform a raw write transaction	*Request:* • `u16` slave address • `u8[]` bytes to write
1	**READ** Perform a 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 transaction. Multiple registers can be written at once if the slave supports 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 at once if the slave supports auto-increment.	*Request:* • `u16` slave address • `u8` register number • `u16` number of read bytes *Response:* • `u8[]` received bytes

15.8 USART Unit

The USART unit provides access to one of the microcontroller's USART peripherals. See Section 7.1 for more information about the interface.

Most USART parameters available in the hardware peripheral's configuration registers can be adjusted to match the application's needs. The peripheral is capable of driving RS-485 transceivers, using the Driver Enable (DE) output for switching between reception and transmission.

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

add a diagram of the dma-based reception

15.8.1 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
```

15.8.2 USART Events

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

15.8.3 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

15.9 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 ROM codes 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 ROM codes, unique 64-bit (8-byte) identifiers that work as addresses. When only one device is connected, the value 0 may be used instead and the addressing will be skipped. Its ROM code may be recovered using the READ_ADDR command or by the search algorithm.

15.9.1 1-Wire Configuration

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

15.9.2 1-Wire Commands

Code	Function	Structure
0	**CHECK_PRESENCE** Test if there are any devices attached to the bus.	*Response:* • `bool` presence detected
1	**SEARCH_ADDR** Start the search algorithm.	*Response:* • `bool` should continue • `u64[]` ROM codes
2	**SEARCH_ALARM** Start the search algorithm, finding only devices in an alarm state.	*Response:* • `bool` should continue • `u64[]` ROM codes
3	**SEARCH_CONTINUE** Continue a previously started search	*Response:* • `bool` should continue • `u64[]` ROM codes
4	**READ_ADDR** Read a device address (single device only)	*Response:* • `u64` ROM code
10	**WRITE** Write bytes to a device.	*Request:* • `u64` ROM code • `u8[]` bytes to write

Code	Function	Structure
11	**READ** Write a request and read response.	*Request:* • `u64` ROM code • `u16` read length • `bool` verify checksum • `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.	

15.10 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 optional averaging. Continuous mode does not support averaging, but the latest measurement can be read at any time without a delay.

15.10.1 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 Arm Cortex-M, 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 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):

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

15.10.2 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
```

15.10.3 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	

Code	Function	Structure
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
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

Code	Function	Structure
20	**SET_POLARITY** Set pulse polarity (active level)	*Response:* • `bool` polarity
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	

15.11 ADC Unit

The analog/digital converter unit is one of the most complicated and powerful 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 is 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, a block capture which continues indefinitely, until the host decides to stop the stream.

It is possible to activate any number of the 16 analog inputs of the ADC peripheral simultaneously, together with the internal input channels. 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 host increases. Those high speeds are achievable in shorter block captures, taking advantage of the (configurable) data buffer. A streamed or too long block capture may be aborted after the buffer is exhausted.

add a diagram

15.11.1 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
```

15.11.2 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
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

15.11.3 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:* • `float[]` smoothed values 0–4095
2	**READ_CAL_CONSTANTS** Read factory calibration constants from the MCU's ROM	*Response:* • `u16` V_{REF_INT} voltage (raw ADC word) • `u16` ADC reference voltage (mV) during V_{REF_INT} measurement • `u16` Temperature sensor voltage in point 1 (raw ADC word) • `u16` Temperature sensor voltage in point 2 (raw ADC word) • `u16` Temperature in point 1 (°C) • `u16` Temperature in point 2 (°C) • `u16` ADC reference voltage (mV) during temp. sensor calibration
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 • `float` achieved sample rate
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)

Code	Function	Structure
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

15.12 DAC Unit

The digital/analog unit works with the two-channel DAC hardware peripheral. 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, can be synchronized, and their phase offset and frequency are dynamically adjustable.

15.12.1 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
```

15.12.2 DAC Commands

Channels are specified in all commands as a 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 sine waveform	*Request:* • u8 channels
2	**WAVE_TRIANGLE** Start a symmetrical triangle waveform	*Request:* • u8 channels
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)

Code	Function	Structure
10	**SYNC** Synchronize the two channels. The phase accumulator is reset to zero.	
20	**SET_FREQUENCY** Set the channel frequency	*Request:* • `u8` channels • `float` frequency
21	**SET_PHASE** Set a channel's phase. It is recommended to only set the phase 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)

15.13 PWM Unit

The PWM unit uses a timer/counter to generate a PWM (pulse train) signal. There are four outputs with a common frequency and independent duty cycles. Each channel can be individually enabled or disabled. This unit is intended for applications such as light dimming, heater regulation, or the control of H-bridges.

15.13.1 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
```

15.13.2 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.	

15.14 Touch Sense Unit

The touch sensing unit provides an access to the TSC peripheral, explained in Section 8.4. The unit configures the TSC and reads the output values of each enabled touch pad. Additionally, a threshold-based digital input function is implemented to make the emulation of touch buttons easier. The hysteresis and debounce time can be configured in the configuration file or set using a command. The threshold of individual pads must be set using a command.

15.14.1 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=
# ...
```

15.14.2 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

15.14.3 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.	

Code	Function	Structure
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

Chapter 16

Client Software

With the communication protocol clearly defined in Chapters 12 and 15, respective Chapter 13 for the wireless gateway, the implementation of a client software is relatively straightforward. Two client libraries have been developed, in languages C and Python.

16.1 General Library Structure

The structure of a GEX client library is in all cases similar:

- **USB or serial port access**

  This is the only platform-dependent part of the library. Unix-based systems provide a standardized POSIX API to configure the serial port. A raw access to USB endpoints is possible using the libUSB C library. Access to the serial port or USB from C on MS Windows has not been investigated, but should be possible using proprietary APIs.

  Accessing the serial port or USB endpoints from Python is more straightforward thanks to the cross platform libraries *PySerial* and *PyUSB*.

- **TinyFrame implementation**

  The *TinyFrame* protocol library can be used directly in desktop C applications, and it has been re-implemented in Python and other languages.

- **Higher-level GEX logic**

  The host side of the communication protocol described in Chapter 12 should be implemented as a part of the library. This includes the reading and writing of configuration files, unit list read-out, command payload building, and asynchronous event parsing.

  Additional utilities may be defined on top of this basic protocol support for the command API of different GEX units, as described in Chapter 15. Those unit-specific "drivers" are available in the provided Python library.

16.2 Python Library

The Python GEX library implements both a serial port access and a raw access to USB endpoints. Its development has been prioritized over the C library because of its potential

to integrate with MATLAB, and because it promises to be the most convenient method to interact with GEX thanks to the ease-of-use that comes with the Python syntax. This library provides a high level API above the individual unit types, removing the burden of building and parsing of the binary command payloads from the user.

The library is composed of a *transport* class, the core class `gex.Client`, and unit classes (e.g., `gex.I2C` or `gex.SPI`).

Three transport implementations have been developed:

- `gex.TrxSerialSync` – virtual serial port access with polling for a response
- `gex.TrxSerialThread` – virtual serial port access with a polling thread and semaphore-based notifications
- `gex.TrxRawUSB` – similar to `gex.TrxSerialThread`, but using a raw USB endpoint access

The wireless gateway is accessed by wrapping either of the transports in an instance of `gex.DongleAdapter` before passing it to `gex.Client`.

16.2.1 Example Python Script

An example Python program displaying a test pattern on a LED matrix using the I^2C-connected driver chip IS31FL3730 is presented in Listing 3 as an illustration of the library usage. A photo of the produced LED pattern can be seen in Figure 16.1.

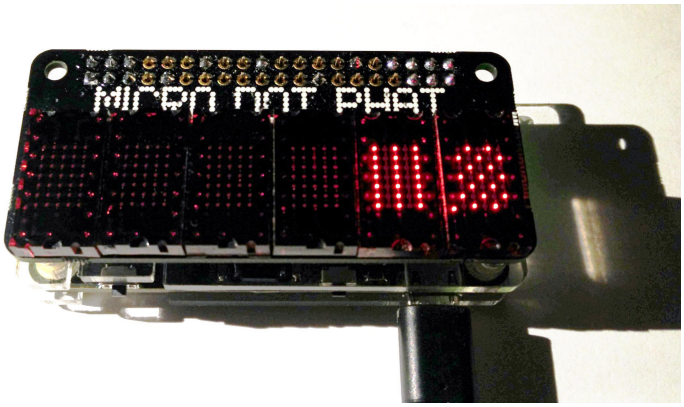


Figure 16.1: GEX Zero with the Micro Dot pHAT add-on board, showing a test pattern defined in a Python script

First, a client instance is created, receiving the transport as an argument. We use a With block in the example to ensure the transport is safely closed before the program ends, even if that happens due to an exception; this is similar to the Try-Finally pattern in Java. The client (and subsequently the transport) can be closed manually by calling its `.close()` method. Inside the With block, the script proceeds to create unit handles and use them to perform the desired task, in our case a communication with the LED matrix driver over the I^2C bus.

```python
#!/bin/env python3
# The I2C unit, called 'i2c', is configured to use PB6 and PB7
import gex
with gex.Client(gex.TrxRawUSB()) as client:
bus = gex.I2C(client, 'i2c')
addr = 0x61
bus.write_reg(addr, 0x00, 0b00011000) # dual matrix
bus.write_reg(addr, 0x0D, 0b00001110) # 34 mA
bus.write_reg(addr, 0x19, 64) # set brightness
# matrix 1
bus.write_reg(addr, 0x01, [
0xAA, 0x55, 0xAA, 0x55,
0xAA, 0x55, 0xAA, 0x55
])
# matrix 2
bus.write_reg(addr, 0x0E, [
0xFF, 0x00, 0xFF, 0x00,
0xFF, 0x00, 0xFF, 0x00
])
# update display
bus.write_reg(addr, 0x0C, 0x01)
```

Listing 3: An example Python program using the GEX client library

16.3 MATLAB integration

The Python library can be accessed from MATLAB scripts thanks to the MATLAB's two-way Python integration [58]. Controlling GEX from MATLAB may be useful when additional processing is required, e.g., with data from the ADC; however, in many cases, an open source alternative native to Python exists that could be used for the same purpose, such as the NumPy and SciPy libraries [59].

The example in Listing 4 demonstrates the use of MATLAB to calculate the frequency spectrum of an analog signal captured with GEX. The syntax needed to use the serial port transport (instead of a raw access to USB endpoints) is shown in a comment.

16.4 C Library

The C library is more simplistic than the Python one; it supports only the serial port transport (UART or CDC/ACM) and does not implement asynchronous polling or the unit support drivers. What *is* implement—the transport, a basic protocol handler, and payload building and parsing utilities—is sufficient for most applications, though less convenient than the Python library.

```
% The ADC unit, called 'adc', is configured to use PA1 as Channel 0

%trx = py.gex.TrxSerialThread(pyargs('port', '/dev/ttyUSB1', ...
%                                    'baud', 115200));
trx = py.gex.TrxRawUSB();
client = py.gex.Client(trx);
adc = py.gex.ADC(client, 'adc');

L = 1000;
Fs = 1000;
adc.set_sample_rate(uint32(Fs)); % casting to unsigned integer
data = adc.capture(uint32(L));
data = double(py.array.array('f',data)); % numpy array to matlab format

Y = fft(data);
P2 = abs(Y/L);
P1 = P2(1:L/2+1);
P1(2:end-1) = 2*P1(2:end-1);
f = Fs*(0:(L/2))/L;

plot(f,P1)
client.close()
```

Listing 4: Calling the Python GEX library from a MATLAB script

This low-level library is intended for applications where the performance of the Python implementation is insufficient, or where an integration with existing C code is required. The full API can be found in the library header files. A C version of the example Python script shown above, controlling a LED matrix driver, is presented in Listing 5. Readers might point out that this example is unnecessarily obtuse, and that the payloads could be constructed in a more readable way. Indeed, two better methods of payload construction are available: one using C structs, and the other taking advantage of the Payload Builder utility bundled with TinyFrame, which is included in the library package.

16.4.1 Structure-based Payload Construction

The structure-based method utilizes C structs to access individual fields in the payload. Simple payloads can be represented by a struct without problems, but payloads of a dynamic length pose a challenge; we can either define a new struct for each required length, or, when the variable-length array is located at the end of the payload, a struct with the largest needed payload size is defined and the real length is then specified when sending the message. The latter approach is illustrated in Listing 6.

16.4.2 Using the Payload Builder Utility

```c
#include <signal.h>
#include <assert.h>
#include "gex.h"

void main(void)
{
    // Initialize GEX and the I2C unit handle
    GexClient *gex = GEX_Init("/dev/ttyACM0", 200);
    assert(NULL != gex);
    GexUnit *bus = GEX_GetUnit(gex, "i2c", "I2C");
    assert(NULL != bus);

    // Configure the matrix driver
    GEX_Send(bus, 2, (uint8_t*) "\x61\x00\x00\x18", 4);
    GEX_Send(bus, 2, (uint8_t*) "\x61\x00\x0d\x0e", 4);
    GEX_Send(bus, 2, (uint8_t*) "\x61\x00\x19\x40", 4);

    // Load data
    GEX_Send(bus, 2, (uint8_t*) "\x61\x00\x01"
                                "\xAA\x55\xAA\x55\xAA\x55\xAA\x55", 11);
    GEX_Send(bus, 2, (uint8_t*) "\x61\x00\x0e"
                                "\xFF\x00\xFF\x00\xFF\x00\xFF\x00", 11);
    // Update display
    GEX_Send(bus, 2, (uint8_t*) "\x61\x00\x0c\x01", 4);

    GEX_DeInit(gex);
}
```

Listing 5: An example C program (GNU C99) controlling GEX using the low-level GEX library; this code has the same effect as the Python script shown in Listing 3, with payloads built following the command tables from Chapter 15.

The Payload Builder utility offers a flexible solution to the construction of arbitrary binary payloads. It is used in the GEX firmware to construct messages and events, along with the binary settings storage content.

An example of Payload Builder's usage is shown in Listing 7. We give it a byte buffer and it then fills it with the payload values, taking care of buffer overflow and to advance the write pointer by the right number of bytes. The third parameter of `pb_init()` is optional, a pointer to a function called when the buffer overflows; this callback can flush the buffer and rewind it, or report an error.

Payload Builder is accompanied by Payload Parser, a tool doing the exact opposite. While it is not needed in our example, we will find this utility useful when processing command responses or events payloads. The full API of those utilities can be found in their header files.

```
struct i2c_write {
        uint16_t address;
        uint8_t reg;
        uint8_t value[8]; // largest needed payload size
} __attribute__((packed));

// 1-byte value
GEX_Send(bus, 2, (void *) &(struct i2c_write) {
        .address = 0x61,
        .reg = 0x00,
        .value = {0x18},
}, 3 + 1); // use only 1 byte of the value array

// 8-byte value
GEX_Send(bus, 2, (void *) &(struct i2c_write) {
        .address = 0x61,
        .reg = 0x01,
        .value = {0xAA, 0x55, 0xAA, 0x55, 0xAA, 0x55, 0xAA, 0x55},
}, 3 + 8);
```

Listing 6: The variable-length struct approach to payload building

```
uint8_t buff[20];
PayloadBuilder pb;

pb = pb_init(&buff, 20, NULL);
pb_u16(&pb, 0x61);
pb_u8(&pb, 0x00);
pb_u8(&pb, 0x18);
GEX_Send(bus, 2, buff, pb_length(&pb));

pb_rewind(&pb); // reset the builder for a new frame

uint8_t screen[8] = {0xAA, 0x55, 0xAA, 0x55, 0xAA, 0x55, 0xAA, 0x55};
pb_u16(&pb, 0x61);
pb_u8(&pb, 0x01);
pb_buf(&pb, &screen, 8);
GEX_Send(bus, 2, buff, pb_length(&pb));
```

Listing 7: Building and sending payloads using the PayloadBuilder utility

Part IV

Results

Chapter 17

Conclusion

TODO

Bibliography

[1] Seeed Technology Co.,Ltd. *Bus Pirate v4 product page*. URL: https://www.seeedstudio.com/Bus-Pirate-v4-p-740.html (visited on 05/12/2018).

[2] National Instruments. *I²C/SPI Interface Device product page*. URL: https://www.ni.com/en-gb/shop/select/i2c-spi-interface-device (visited on 05/12/2018).

[3] National Instruments. *USB-6008 product page*. URL: http://www.ni.com/en-gb/support/model.usb-6008.html (visited on 05/12/2018).

[4] Total Phase, Inc. *USB-6008 product page*. URL: https://www.totalphase.com/products/aardvark-i2cspi/ (visited on 05/12/2018).

[5] USB Implementers Forum, Inc. *Universal Serial Bus Specification*. 2000. URL: http://www.usb.org/developers/docs/usb20_docs/ (visited on 05/12/2018).

[6] Craig Peacock. *USB in a NutShell*. URL: https://www.beyondlogic.org/usbnutshell (visited on 05/12/2018).

[7] MQP Electronics Ltd. *USB Made Simple*. 2008. URL: http://www.usbmadesimple.co.uk/ (visited on 05/12/2018).

[8] Microsoft Corporation. *Windows 2000 Professional Resource Kit / USB Functions*. 2008. URL: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-2000-server/cc939102(v%3dtechnet.10) (visited on 05/12/2018).

[9] USB Implementers Forum, Inc. *USB Class Codes*. 2016. URL: http://www.usb.org/developers/defined_class (visited on 05/12/2018).

[10] *pid.codes, a registry of USB PID codes for open source hardware projects*. URL: http://pid.codes/ (visited on 05/12/2018).

[11] EEVblog Electronics Community Forum. *pid.codes, a registry of USB PID codes for open source hardware projects*. URL: https://www.eevblog.com/forum/projects/driving-the-1k5-usb-pull-up-resistor-on-d/ (visited on 05/12/2018).

[12] USB Implementers Forum, Inc. *USB Mass Storage Class, Specification Overview*. 2010. URL: http://www.usb.org/developers/docs/devclass_docs/Mass_Storage_Specification_Overview_v1.4_2-19-2010.pdf (visited on 05/12/2018).

[13] USB Implementers Forum, Inc. *USB Mass Storage Class, Bulk-Only Transport*. 1999. URL: http://www.usb.org/developers/docs/devclass_docs/usbmassbulk_10.pdf (visited on 05/12/2018).

[14] *Discussion in a USB storage driver development mailing list (site defunct, archived via Archive.org)*. 2004. URL: https://web.archive.org/web/20071108121822/https://lists.one-eyed-alien.net/pipermail/usb-storage/2004-September/000795.html (visited on 05/12/2018).

[15] Jan Axelson. *Mass Storage FAQ*. 2013. URL: http://janaxelson.com/mass_storage_faq.htm (visited on 05/12/2018).

[16] USB Implementers Forum, Inc. *Class definitions for Communication Devices 1.2*. 2010. URL: http://www.usb.org/developers/docs/devclass_docs/CDC1.2_WMC1.1_012011.zip (visited on 05/12/2018).

[17] USB Implementers Forum, Inc. *USB Interface Association Descriptor, Device Class Code and Use Model*. 2003. URL: http://www.usb.org/developers/docs/whitepapers/iadclasscode_r10.pdf (visited on 05/12/2018).

[18] Real Time Engineers Ltd. *FreeRTOS Ports*. URL: https://www.freertos.org/a00090.html (visited on 05/12/2018).

[19] Real Time Engineers Ltd. *The FreeRTOS™ Reference Manual*. Real Time Engineers Ltd., 2018. URL: https://www.freertos.org/Documentation/FreeRTOS_Reference_Manual_V10.0.0.pdf (visited on 05/12/2018).

[20] Richard Barry. *Mastering the FreeRTOS™ Real Time Kernel. A Hands-On Tutorial Guide*. Real Time Engineers Ltd., 2016. URL: https://www.freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf (visited on 05/12/2018).

[21] Real Time Engineers Ltd. *How FreeRTOS Works: FreeRTOS Implementation*. URL: https://www.freertos.org/implementation/main.html (visited on 05/12/2018).

[22] Wikipedia contributors. *Comparison of File Systems / OS Support*. URL: https://en.wikipedia.org/wiki/Comparison_of_file_systems#OS_support (visited on 05/12/2018).

[23] Microsoft Corporation. *How FAT Works*. 2009. URL: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc776720(v=ws.10) (visited on 05/12/2018).

[24] Jack Dobiash. *FAT16 Structure Information*. 1999. URL: http://home.teleport.com/~brainy/fat16.htm (visited on 05/12/2018).

[25] LKT Software. *FAT16 File System*. 1999. URL: http://www.maverick-os.dk/FileSystemFormats/FAT16_FileSystem.html (visited on 05/12/2018).

[26] Bob Eager. *A tutorial on the FAT file system*. 2017. URL: http://www.tavi.co.uk/phobos/fat.html (visited on 05/12/2018).

[27] Microsoft Corporation. *FAT: General Overview of On-Disk Format*. Tech. rep. 2000. URL: https://staff.washington.edu/dittrich/misc/fatgen103.pdf (visited on 05/12/2018).

[28] "vinDaci". *Long Filename Specification*. 1998. URL: http://home.teleport.com/~brainy/lfn.htm (visited on 05/12/2018).

[29] Arm Mbed. *Arm Mbed DAPLink source code repository*. 2018. URL: https://github.com/ARMmbed/DAPLink (visited on 05/12/2018).

[30] "nkcelectronics". *Retrofitting AutoReset feature into an old Arduino serial board*. URL: https://playground.arduino.cc/Learning/AutoResetRetrofit (visited on 05/13/2018).

[31] SD Group. *SD Specifications, Part 1: Physical Layer Simplified Specification*. Tech. rep. 2010. URL: https://www.cs.utexas.edu/~simon/395t_os/resources/Part_1_Physical_Layer_Simplified_Specification_Ver_3.01_Final_100518.pdf (visited on 05/13/2018).

[32] NXP Semiconductors. *I2C-bus specification and user manual*. 2014. URL: https://www.nxp.com/docs/en/user-guide/UM10204.pdf (visited on 05/12/2018).

[33] Jared Becker Jonathan Valdez. *Understanding the I2C Bus*. Tech. rep. 2015. URL: http://www.ti.com/lit/an/slva704/slva704.pdf (visited on 05/12/2018).

[34] Dallas Semiconductor. *AN214: Using a UART to Implement a 1-Wire Bus Master*. Tech. rep. URL: https://www.maximintegrated.com/en/app-notes/index.mvp/id/214 (visited on 05/13/2018).

[35] Dallas Semiconductor. *DS18S20 High Precision 1-Wire Digital Thermometer*. Tech. rep. URL: https://datasheets.maximintegrated.com/en/ds/DS18S20.pdf (visited on 05/13/2018).

[36] Dallas Semiconductor. *AN162: Interfacing the DS18X20/DS1822 1-Wire Temperature Sensor in a Micro-controller Environment*. Tech. rep. URL: https://www.maximintegrated.com/en/app-notes/index.mvp/id/162 (visited on 05/13/2018).

[37] Worldsemi. *WS2812B datasheet*. Tech. rep. URL: www.world-semi.com/DownLoadFile/108 (visited on 05/13/2018).

[38] SiTime Corporation. *AN10033: Frequency Measurement Guidelines for Oscillators*. Tech. rep. URL: https://www.sitime.com/api/gated/AN10033-Frequency-Measurement-Guidelines-for-Oscillators.pdf (visited on 05/13/2018).

[39] Paul Boven. "Increasing the resolution of reciprocal frequency counters". In: *Proceedings of the 50. VHF meeting in Weinheim*. URL: https://www.febo.com/pipermail/time-nuts/attachments/20071201/e7833af5/attachment.pdf (visited on 05/13/2018).

[40] Maxim Integrated. *AN1080: Understanding SAR ADCs: Their Architecture and Comparison with Other ADCs*. Tech. rep. 2001. URL: https://pdfserv.maximintegrated.com/en/an/AN1080.pdf (visited on 05/13/2018).

[41] Eva Murphy and Colm Slattery. "All about direct digital synthesis". In: Ask The Application Engineer 33 (2004). URL: http://www.analog.com/media/en/analog-dialogue/volume-38/number-3/articles/all-about-direct-digital-synthesis.pdf.

[42] ST Microelectronics. *RM0091: STM32F0x1/STM32F0x2/STM32F0x8 reference manual*. 2017. URL: http://www.st.com/resource/en/reference_manual/dm00031936.pdf (visited on 05/12/2018).

[43] ST Microelectronics. *STM32L4 training: Touch Sensing Controller*. Tech. rep. 2017. URL: http://www.st.com/resource/en/product_training/stm32l4_peripheral_touchsense.pdf (visited on 05/12/2018).

[44] ST Microelectronics. *Touch Sensing Controller (TSC) presentation*. Tech. rep. 2015. URL: https://wenku.baidu.com/view/8472044a6137ee06eef9180c.html?re=view (visited on 05/12/2018).

[45] ST Microelectronics. *AN4299: Guidelines to improve conducted noise robustness on STM32F0 Series, STM32F3 Series, STM32L0 Series and STM32L4 Series touch sensing applications*. Tech. rep. 2018. URL: http://www.st.com/resource/en/application_note/dm00085385.pdf (visited on 05/12/2018).

[46] ST Microelectronics. *AN4310: Sampling capacitor selection guide for MCU based touch sensing applications*. Tech. rep. 2015. URL: http://www.st.com/resource/en/application_note/dm00087593.pdf (visited on 05/12/2018).

[47] ST Microelectronics. *AN4312: Guidelines for designing touch sensing applications with surface sensors*. Tech. rep. 2017. URL: http://www.st.com/resource/en/application_note/dm00087990.pdf (visited on 05/12/2018).

[48] ST Microelectronics. *AN4316: Tuning a STMTouch-based application*. Tech. rep. 2015. URL: http://www.st.com/resource/en/application_note/dm00088471.pdf (visited on 05/12/2018).

[49] USB Implementers Forum, Inc. *USB Device Class Specification for Device Firmware Upgrade*. 2004. URL: http://www.usb.org/developers/docs/devclass_docs/DFU_1.1.pdf (visited on 05/17/2018).

[50] Harald Welte and Stefan Schmidt and Tormod Volden. *dfu-util*. 2016. URL: http://dfu-util.sourceforge.net/ (visited on 05/17/2018).

[51] ST Microelectronics. *STSW-STM32102: STM32 Virtual COM Port Driver*. URL: http://www.st.com/en/development-tools/stsw-stm32102.html (visited on 05/17/2018).

[52] ST Microelectronics. *STM32Cube initialization code generator*. URL: http://www.st.com/en/development-tools/stm32cubemx.html (visited on 05/18/2018).

[53] Ondřej Hruška. *TinyFrame, a library for building and parsing data frames for serial interfaces*. URL: https://github.com/MightyPork/TinyFrame (visited on 05/13/2018).

[54] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (Aug. 2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[55] Semtech Corporation. *SX1276/77/78/79 datasheet*. 2016. URL: https://www.semtech.com/uploads/documents/DS_SX1276-7-8-9_W_APP_V5.pdf (visited on 05/12/2018).

[56] Nordic Semiconductor. *nRF24L01+ Single Chip 2.4GHz Transceiver Product Specification v1.0*. 2008. URL: http://www.nordicsemi.com/eng/content/download/2726/34069/file/nRF24L01P_Product_Specification_1_0.pdf (visited on 05/12/2018).

[57] ST Microelectronics. *Discovery kit with STM32F072RB MCU*. URL: http://www.st.com/en/evaluation-tools/32f072bdiscovery.html (visited on 05/16/2018).

[58] The MathWorks, Inc. *Using MATLAB with Python*. URL: https://www.mathworks.com/solutions/matlab-and-python.html (visited on 05/13/2018).

[59] SciPy developers. *SciPy.org*. URL: https://www.scipy.org/ (visited on 05/13/2018).

Appendices

Schematics here ....