

Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Measurement

Learning and automation GPIO platform

Ondřej Hruška

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

Field of study: Cybernetics and Robotics

Subfield: Sensors and Instrumentation

2018

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.
Supervisor's signature

Head of department's signature

prof. Ing. Pavel Ripka, CSc.
Dean's signature

III. Assignment receipt

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

Date of assignment receipt

Student's signature

Declaration

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

V Praze, 27. května 2018

Acknowledgements

blabla

Abstract

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

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

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

Keywords:

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

Abstrakt

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

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

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

Klíčová slova:

Překlad názvu: Výuková a automatizační GPIO platforma

Contents

Part I Introduction

1 Motivation	3
1.1 The Project's Expected Outcome	4
2 Requirement Analysis	7
2.0.1 Interfacing Intelligent Modules	7
2.0.2 Analog Signal Acquisition	7
2.0.3 Analog Signal Output	8
2.0.4 Logic Level Input and Output	8
2.0.5 Pulse Generation and Measurement	8
2.1 Connection to the Host Computer	8
2.1.1 Messaging	8
2.1.2 Configuration Files.....	9
2.2 Planned Feature List	9
2.3 Microcontroller Selection	10
2.4 Form Factor Considerations	10
3 Existing Solutions	13
3.1 Bus Pirate	13
3.2 Raspberry Pi	14
3.3 Professional DAQ Modules.....	14
3.4 The Firmata Protocol	15

Part II Theoretical Background

4 Universal Serial Bus	19
4.1 Basic Principles and Terminology	20
4.2 USB Physical Layer	23
4.3 USB Classes	23
4.3.1 Mass Storage Class	24
4.3.2 CDC/ACM Class	24
4.3.3 Interface Association: Composite Class	25

5 FreeRTOS	27
5.1 Basic FreeRTOS Concepts and Functions	27
5.1.1 Tasks	27
5.1.2 Synchronization Objects	28
6 The FAT16 Filesystem and Its Emulation	29
7 Supported Hardware Buses	31
7.1 UART and USART	31
7.1.1 Examples of Devices Using UART	31
7.2 SPI	32
7.2.1 Examples of Devices Using SPI	32
7.3 I2C	33
7.3.1 Examples of Devices Using I2C	34
7.4 1-Wire	34
7.4.1 Examples of Devices Using 1-Wire	35
7.5 NeoPixel	35
8 Additional Hardware Functions	39
8.1 Frequency Measurement	39
8.2 Waveform Generation	41
8.2.1 Waveform Generation with DMA and a Timer.....	41
8.2.2 Direct Digital Synthesis	42
Part III	
Firmware Implementation	
9 Application Structure	47
9.1 User's View of GEX.....	47
9.2 Functions of the Core Framework	48
9.3 Resource Allocation	49
9.4 Settings Storage	49
9.5 Functional Blocks	50
9.6 Source Code Layout	51
9.7 Communication Ports	51

9.7.1 USB Connection	52
9.7.2 Communication UART	52
9.7.3 Wireless Connection	52
9.8 Message Passing	52
9.9 Interrupt Routing.....	53
10 Communication Protocol	55
10.1 Frame Structure	55
10.2 Message Listeners.....	56
10.3 Designated Frame Types in GEX	56
10.4 Bulk Read and Write Transactions	57
10.4.1 Bulk Read	57
10.4.2 Bulk Write.....	58
10.4.3 Persisting the Changed Configuration to Flash	58
10.5 Reading a List of Units	58
10.6 Unit Requests and Reports	59
10.6.1 Unit Requests	59
10.6.2 Unit Reports	59
11 Wireless Interface	61
11.1 Comparing SX1276 vs. NRF24L01+	61

Part IV Hardware Design

Appendices

Figures

1.1 A collection of intelligent sensors and devices	3
1.2 An early sketch of a universal bench device	4
2.1 A Discovery board with STM32F072	11
2.2 Form factor sketches	11
3.1 Bus Pirate v.4 (picture by <i>Seeed Studio</i>)	13
3.2 Raspberry Pi 2 (picture by <i>Raspberry Pi Foundation</i>)	14
3.3 Professional tools that GEX can replace	15
4.1 USB hierarchical structure	19
4.2 A detailed view of the host-device connection (<i>USB specification rev. 1.1</i>)	20
4.3 USB descriptors of a GEX prototype obtained using <code>lsusb -vd vid:pid</code>	22
4.4 NRZI encoding example	23
4.5 USB pull-ups	24
7.1 UART frame structure	32
7.2 SPI master with multiple slaves	33
7.3 I2C message diagram	35
7.4 1-Wire topology (by <i>Dallas Semiconductor</i>)	35
7.5 The 1-Wire DIO pulse timing (by <i>Dallas Semiconductor</i>)	36
7.6 A close-up photo of the WS2812B package, showing the LED driver IC	37
7.7 NeoPixel pulse timing diagram and time constraints table (<i>WS2812 datasheet</i>)	37
8.1 Direct frequency measurement method	40
8.2 Reciprocal frequency measurement method	40
8.3 Frequency measurement methods comparison	41
8.4 A simple implementation of the waveform generator	42
8.5 A block diagram of a direct digital synthesis waveform generator	42
9.1 An example allocation in the resource registry	49
9.2 Structure of the settings subsystem	50
9.3 The general structure of the source code repository	51

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 typically involve the generation and sampling of electrical signals coming to and from sensors, actuators, and other circuitry.

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

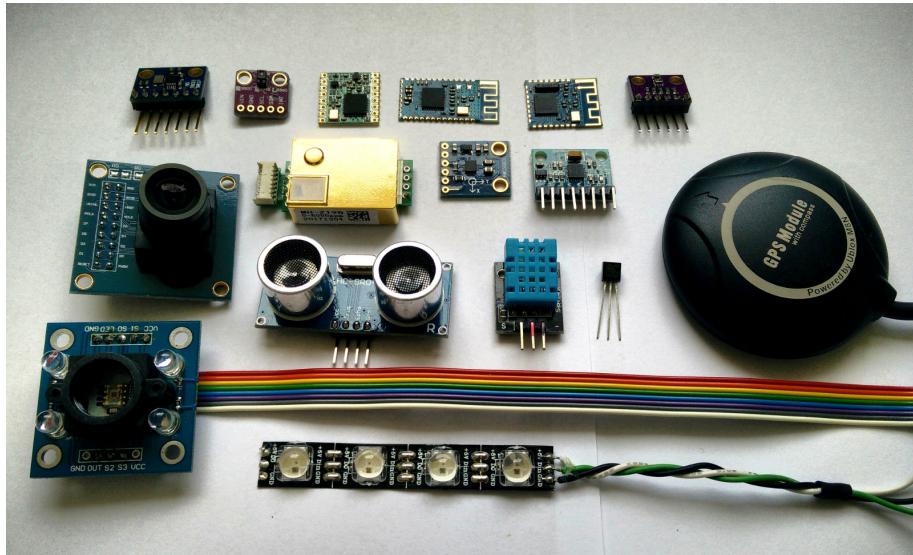


Figure 1.1: A collection of intelligent sensors and devices, most on breadboard adapters: (from top left) a waveform generator, a gesture detector, a LoRa and two Bluetooth modules, an air quality and pressure sensor, a CO₂ sensor, a digital compass, an accelerometer, a GPS module, a camera, an ultrasonic range finder, a humidity sensor, a 1-Wire thermometer, a color detector and an RGB LED strip.

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

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

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

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

1.1 The Project's Expected Outcome

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

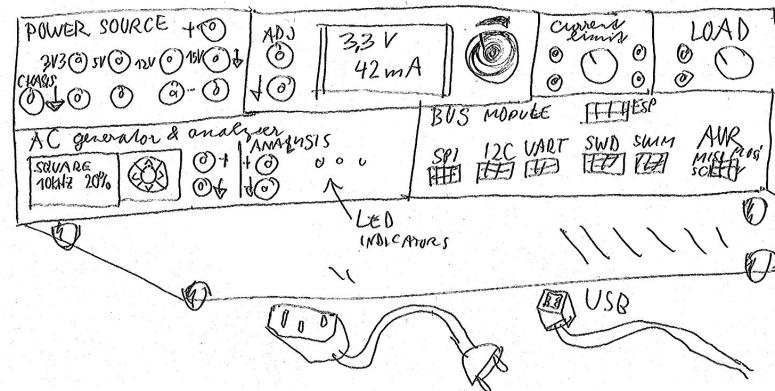


Figure 1.2: An early (2016) sketch of a universal bench device including a power supply, electronic load, a signal generator and a bus module. The bottom half of the panel is in a large part implemented by GEX.

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

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

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

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

Chapter 2

Requirement Analysis

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

2.0.1 Interfacing Intelligent Modules

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

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

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

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

2.0.2 Analog Signal Acquisition

Sometimes it's necessary to use a traditional analog sensor, capture a transient waveform or to just measure a voltage. GEX was meant to focus on digital interfaces, however giving it this capability makes it much more versatile. Nearly all microcontrollers include an analog-digital converter which we can use to measure input voltages and, paired with a timer, to record 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.

■ **2.0.3 Analog Signal Output**

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

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

■ **2.0.4 Logic Level Input and Output**

We've covered some more advanced features, but skipped the simplest feature: a direct access to GPIO pins. Considering the latencies of USB and the PC's operating system, this can't be reliably used for "bit banging", however we can still accomplish a lot with just changing logic levels - e.g. to control character LCDs, or emulate some interfaces that include a clock line, like SPI. As mentioned in [2.0.1](#), 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.

■ **2.0.5 Pulse Generation and Measurement**

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

■ **2.1 Connection to the Host Computer**

■ **2.1.1 Messaging**

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

The most straightforward method of interfacing the board is by passing binary messages in a fashion similar to USART (and plain UART can be available as well). We'll need a duplex connection to enable command confirmations, query-type commands and asynchronous event reporting.

This is possible either using a "Virtual COM port" driver (the CDC/ACM USB class), or through a raw access to the corresponding USB endpoints. Using a raw access avoids potential problems with the operating system's driver interfering or not recognizing the device correctly; on the other hand, having GEX appear as a serial port makes it easier to integrate into existing platforms that have a good serial port support (such as National Instruments LabWindows CVI or MATLAB).

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

[link to where this is better explained](#)

2.1.2 Configuration Files

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

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

2.2 Planned Feature List

Let's list the features we wish to initially support 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
- Dallas 1-Wire
- NeoPixel (addressable LED strips)

- **Communication with the host computer**

- USB connection as virtual serial port or direct endpoint access

- Connection using plain UART
- Wireless attachment

- Configuration

- Fully reconfigurable, temporarily or permanently
- Settings stored in INI files
- File access through the communication API or using a virtual mass storage

2.3 Microcontroller Selection

The STM32F072 microcontroller was chosen for the built prototypes and the initial firmware, owing to it's low cost, advanced peripherals and the availability of development boards. GEX can be later ported to other MCUs, like the STM32L072, STM32F103 or STM32F303.

The STM32F072 is a Cortex M0 device with 128 KiB of flash memory, 16 KiB of RAM and running at 48 MHz. It is equipped with a USB Full Speed peripheral block, a 12-bit ADC and DAC, a number of general purpose timers/counters, SPI, 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.

2.4 Form Factor Considerations

It was mentioned in 1.1 that, while the GEX firmware can be used with existing evaluation boards from ST Microelectronics (figure 2.1), we wish to design and realize a few custom hardware prototypes that will be smaller, more convenient to use and hopefully also cheaper. Three possible form factors are drawn in figure 2.2.

Several factors play a role when deciding what the GEX PCB should look like:

The device must be comfortable and easy to use, which affects the choice of the USB connector, also with respect to cable availability: USB type A is not suitable for desktop computers where it would have to be plugged in the rear of the computer or in the front panel, but it may be usable with laptops; USB Mini-B and Micro-B connectors are both a popular choice in existing kits (e.g. Discovery and Nucleo boards), but Micro-B has a higher rated number of insertions and the cables are ubiquitous thanks to their use in mobile phones, therefore this appears to be the better connector choice.

link to
the in-
sertion
count
spec

The PCB size should be kept minimal to save manufacturing costs. When a standard connector shape and a pin assignment are used we gain the ability to install existing add-on boards designed for other platforms, like the Arduino or Raspberry Pi. Lastly, when the entire board shape is copied from an existing commercial product for which we can buy official or after-market cases, we get an easy access to cases without having to design them ourselves. This is the case of the Raspberry Pi Zero form factor.

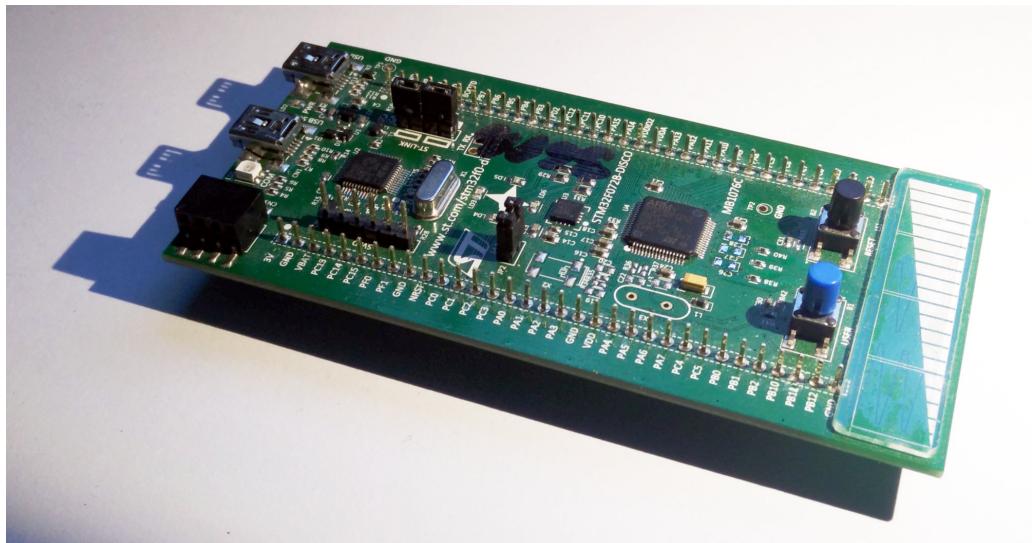


Figure 2.1: A Discovery board with STM32F072 that can be used to run the GEX firmware

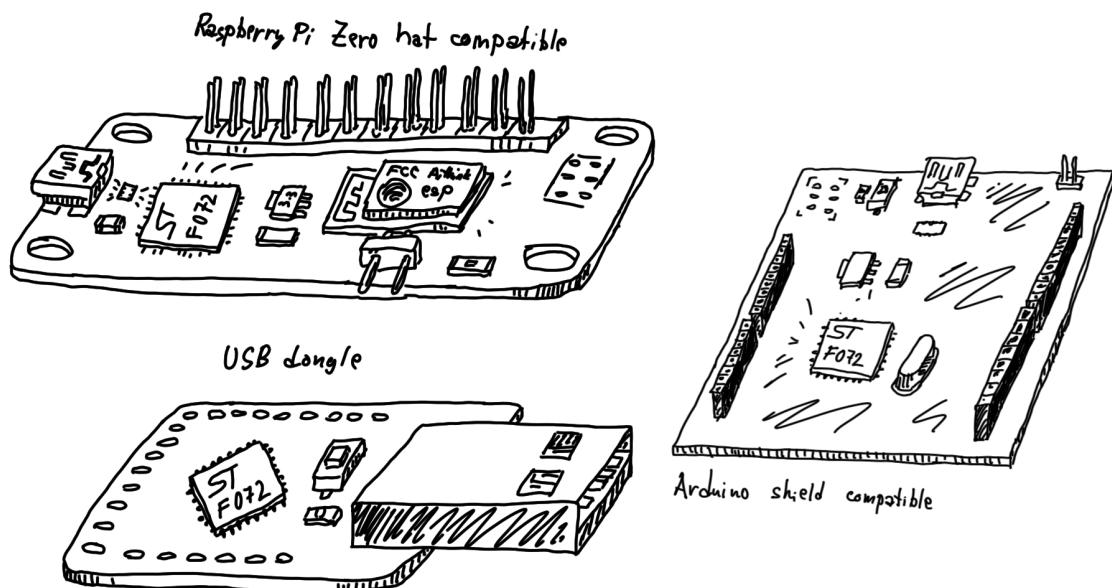


Figure 2.2: A sketch of three possible form factors for the GEX hardware prototype. Note the ESP8266 module which was considered as an option for wireless access but was eventually not used due to its high current usage, unsuitable for battery operation.

Chapter 3

Existing Solutions

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

3.1 Bus Pirate

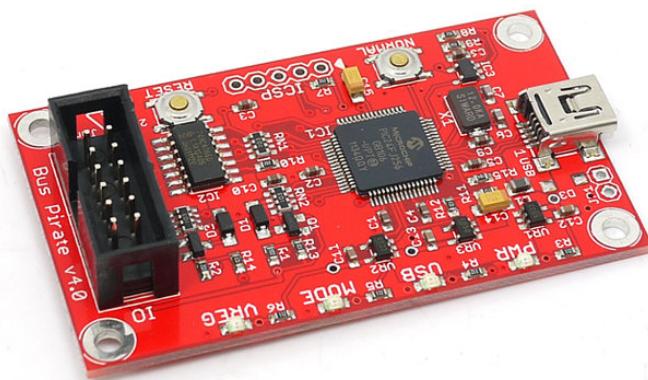


Figure 3.1: Bus Pirate v.4 (picture by *Seeed Studio*)

link to pic source page

Bus Pirate, developed by Ian Lesnet at Dangerous Prototypes and manufactured by Seeed Studio, is a USB-attached device providing access to hardware interfaces like SPI, I²C, USART and 1-Wire, as well as frequency measurement and direct pin access.

[link](#)

[link](#)

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 in scope it's similar to GEX. It can be scripted and controlled from languages like Python or Perl, connects to USB and provides a good selection of hardware interfaces.

The board is based on a PIC16 microcontroller running at 32 MHz. Its analog/digital converter (ADC) only has a resolution of 10 bits (1024 levels). There is no digital/analog

converter (DAC) available on the chip, making applications that require a varied output voltage more difficult. Another limitation of the board is its low number of GPIO pins which may be insufficient for certain applications. The Bus Pirate, at the time of writing, can be purchased for a price similar to some Raspberry Pi models.

3.2 Raspberry Pi

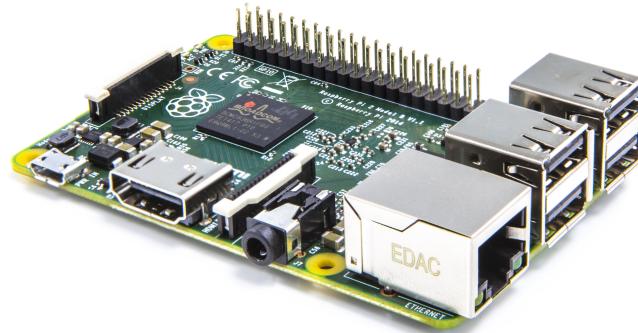


Figure 3.2: Raspberry Pi 2 (picture by *Raspberry Pi Foundation*)

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

The Raspberry Pi is commonly used in schools as a low-cost PC alternative that encourage students' interest in electronics, programming and science. The board is often built into more permanent projects that make use of its powerful processor, such as wildlife camera traps or home automation projects.

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

3.3 Professional DAQ Modules

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

(a) : NI I²C/SPI Interface Device

(b) : NI USB DAQ module

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

Figure 3.3: An example of professional tools that GEX could replace in less demanding scenarios (pictures taken from marketing materials)

3.4 The Firmata Protocol

links

Firmata is a communication protocol based on MIDI (*Musical Instrument Digital Interface*) for passing data to and from embedded microcontrollers. MIDI is mainly used for attaching electronic musical instruments, such as synthesizers, keyboards, mixers etc., to each other or to a PC. Firmata was designed for Arduino as a high level abstraction for its connection to the PC, typically using a FTDI chip or equivalent.

Implementing Firmata in the GEX firmware would make it possible to use existing Firmata libraries on the PC side. However, the protocol is limited by the encompassing MIDI format and isn't very flexible.

Part II

Theoretical Background

Chapter 4

Universal Serial Bus

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

add
those
refs

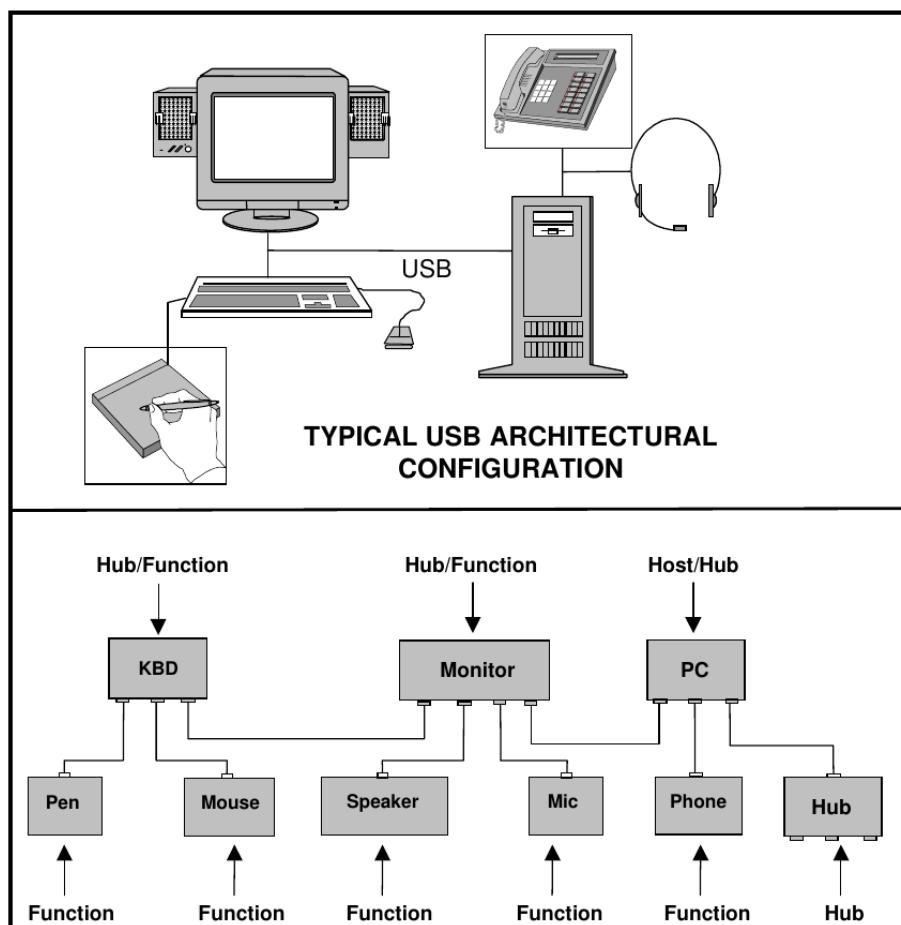


Figure 4.1: A diagram from the USB specification rev. 1.1 showing the hierarchical structure of the USB bus; The PC (Host) controls the bus and initiates all transactions.

4.1 Basic Principles and Terminology

review and correct inaccuracies

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

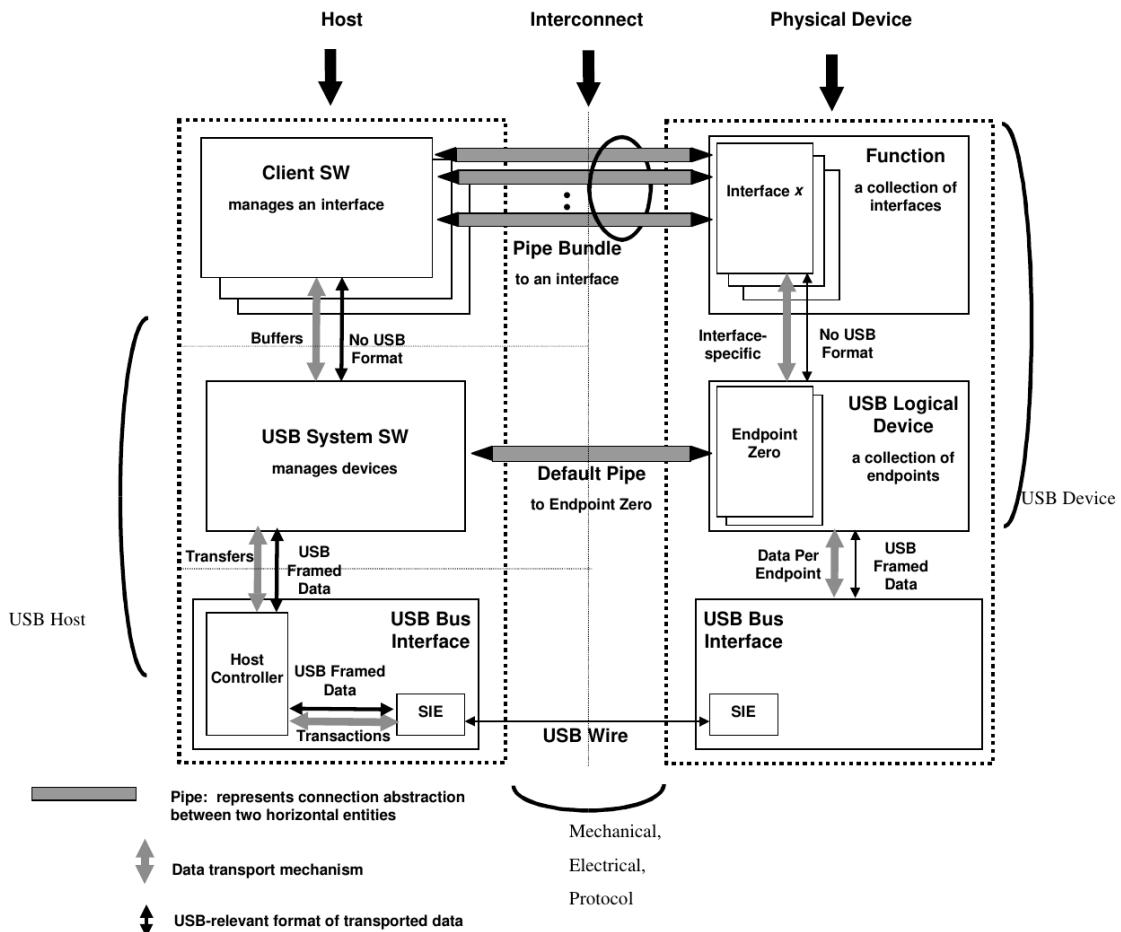


Figure 4.2: A detailed view of the host-device connection (*USB specification rev. 1.1*)

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

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

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

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

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

add reference to the document

4. Universal Serial Bus

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

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

4.2 USB Physical Layer

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

The USB cable contains 4 conductors:

- V_{BUS} (+5 V)
- D+
- D-
- Ground

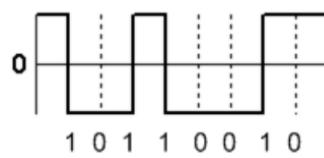


Figure 4.4: NRZI encoding example

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

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

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

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

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

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

4.3 USB Classes

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

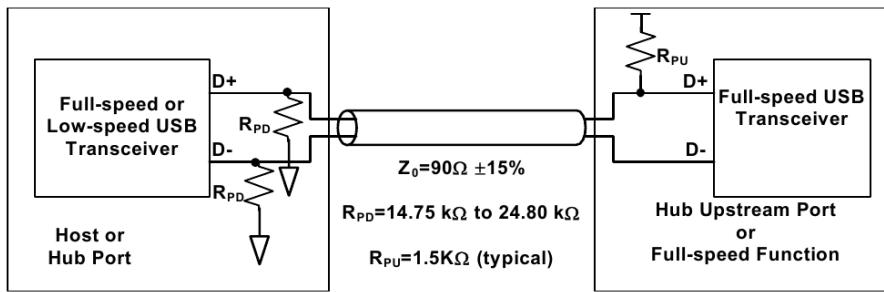


Figure 4.5: Pull-up and pull-down resistors of a Full Speed function, as prescribed by the USB specification rev. 2.0

4.3.1 Mass Storage Class

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

references

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

links

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

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

4.3.2 CDC/ACM Class

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

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

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

verify
this vvv

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

■ 4.3.3 Interface Association: Composite Class

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

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

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

Chapter 5

FreeRTOS

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

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

5.1 Basic FreeRTOS Concepts and Functions

5.1.1 Tasks

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

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

Task Run States

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

■ Task Switching and Interrupts

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

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

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

■ 5.1.2 Synchronization Objects

FreeRTOS provides binary and counting semaphores, mutexes and queues.

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

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

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

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

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

Chapter 6

The FAT16 Filesystem and Its Emulation

...

Chapter 7

Supported Hardware Buses

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

7.1 UART and USART

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

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

USART, as implemented by microcontrollers such as the STM32 family, is a two-wire full duplex interface that uses 3.3 V or 5 V logic levels. The data lines are in the high logical level when idle. A frame, pictured in figure 7.1 starts by a start-bit (low level for the period of one bit) followed by n data bits (typically eight), an optional parity bit and a period of high level called a stop bit or stop bits, usually between one and two bits long.

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

7.1.1 Examples of Devices Using UART

- **MH-Z19B** - NDIR CO₂ concentration sensor

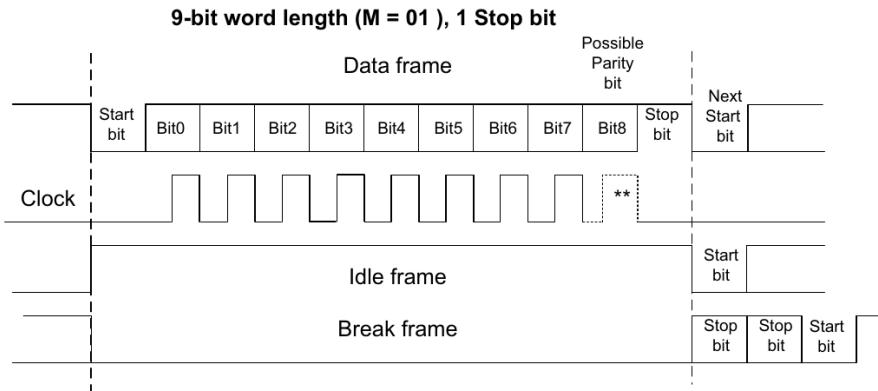


Figure 7.1: UART frame, as shown by the STM32F072 Reference Manual. Break frames are used by some UART based protocols, like LIN (Local Interconnect Network).

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

7.2 SPI

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

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

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

7.2.1 Examples of Devices Using SPI

- **SX1276** - LoRa transceiver
- **nRF24L01+** - 2.4 GHz ISM band radio module

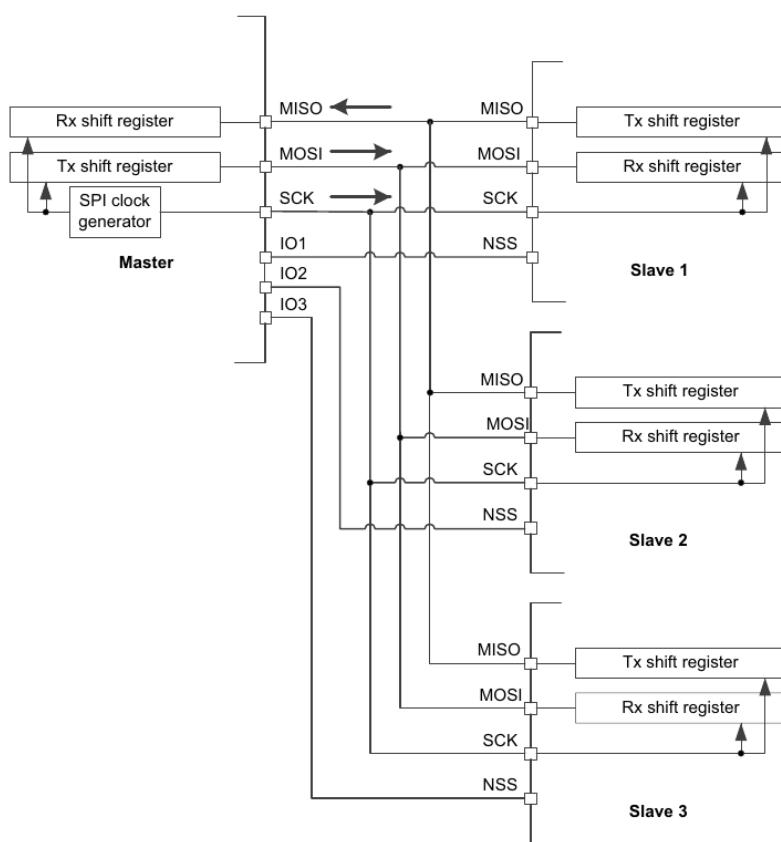


Figure 7.2: A SPI bus with 1 master and 3 slaves, each enabled by its own Slave Select signal (*STM32F072 Reference Manual*)

- **L3GD20** - 3-axis gyroscope
- **BMP280** - pressure sensor
- **BME680** - air quality sensor
- **ENC28J60** - Ethernet controller
- **L6470** - intelligent stepper motor driver
- **AD9833** - DDS-based DAC / waveform generator (MOSI only)
- **ADE7912** - triple $\Sigma\Delta$ ADC for power metering applications
- **SD cards**
- SPI-interfaced EEPROMs and Flash memories

7.3 I2C

I2C is a two-wire (SDA—*Serial Data*, SCL—*Serial Clock*), open-drain bus that supports multi-master operation. The protocol was developed by Philips Semiconductor (now NXP).

Semiconductors) and until 2006 implementors were required to pay licensing fees, leading to the development of compatible implementations with different names, such as Atmel's Two Wire Interface (TWI) or Dallas Semiconductor's "Serial 2-wire Interface" (e.g. used in the DS1307 RTC chip). I2C is the basis of the SMBus and PMBus protocols which add additional constraints and rules for a more robust operation.

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

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

7.3.1 Examples of Devices Using I2C

- **APDS-9960** - ambient light, proximity and gesture sensor
- **L3GD20, BMP280, BME680** - listed as SPI devices, those also support I2C
- **DS1307** - RTC; I2C is not mentioned in the entire datasheet, presumably to avoid paying license fees, but it is fully compatible
- **IS31FL3730** - LED matrix driver
- Cameras with an SCCB port can be accessed through I2C

7.4 1-Wire

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

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

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

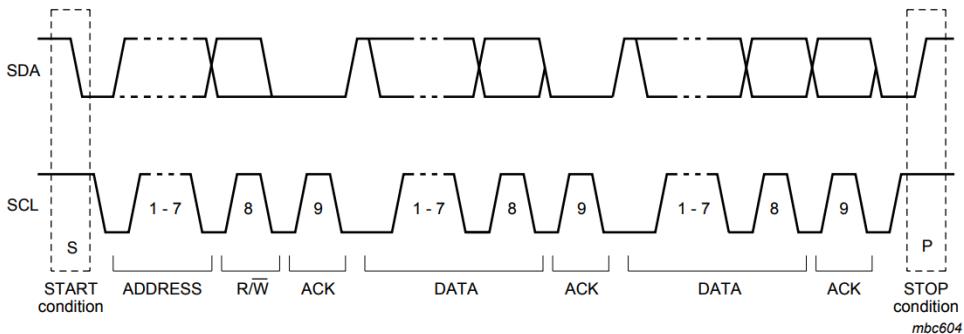


Figure 7.3: An I2C message diagram. The frame starts with a start condition and stops with a stop condition, defined by an SDA edge while SCL is high. The address and data bytes are acknowledged by the slave by sending a 0 on the open-drain SDA line in the following clock cycle. A slave can terminate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

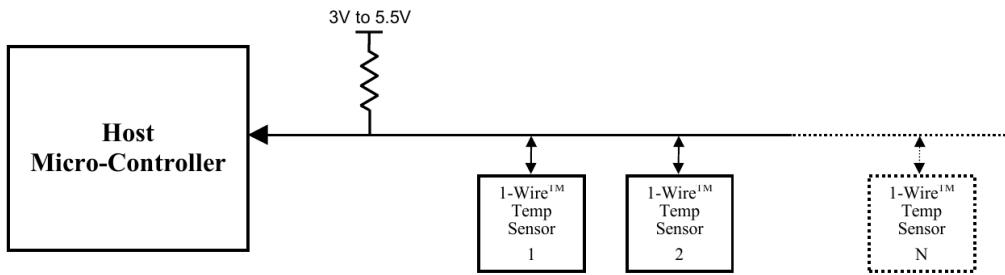


Figure 7.4: 1-Wire topology (by Dallas Semiconductor)

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

The NeoPixel protocol is unidirectional, using only one data pin. The LED drivers are chained together. Ones and zeros are encoded by a pulse length on the data pin; after loading the color data to the LED string, a longer "reset" pulse is issued by the bus master

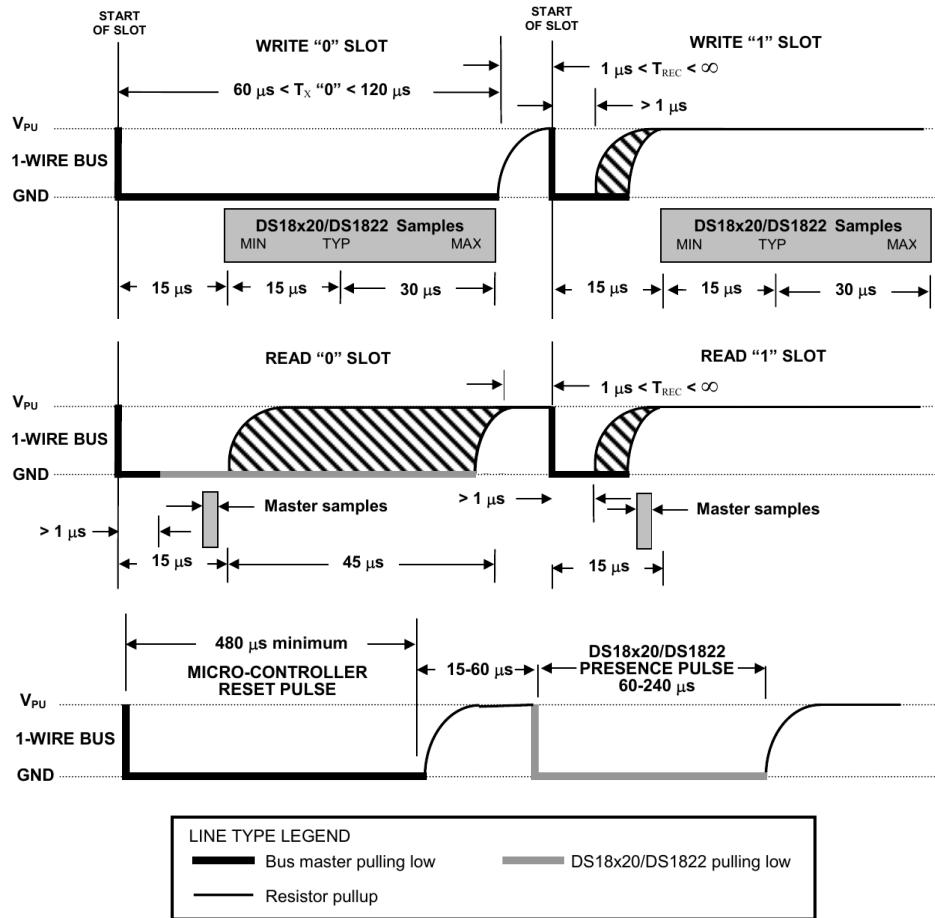


Figure 7.5: The 1-Wire DIO pulse timing (by Dallas Semiconductor)

and the set colors are displayed. The timing diagram and constraints are shown in figure 7.7.

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

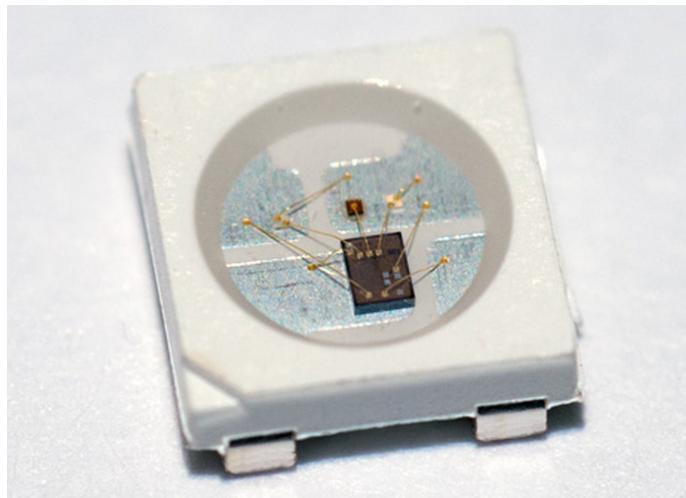
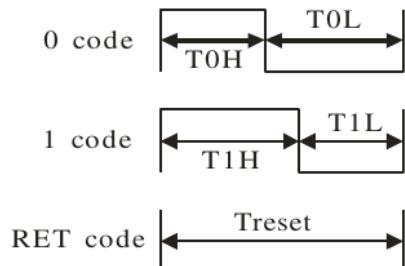


Figure 7.6: A close-up photo of the WS2812B package, showing the LED driver IC



Data transfer time(TH+TL=1.25μs±600ns)

T0H	0 code ,high voltage time	0.4us	±150ns
T1H	1 code ,high voltage time	0.8us	±150ns
T0L	0 code , low voltage time	0.85us	±150ns
T1L	1 code ,low voltage time	0.45us	±150ns
RES	low voltage time	Above 50μs	

Figure 7.7: NeoPixel pulse timing diagram and time constraints table (*WS2812 datasheet*)

Chapter 8

Additional Hardware Functions

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

8.1 Frequency Measurement

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

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

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

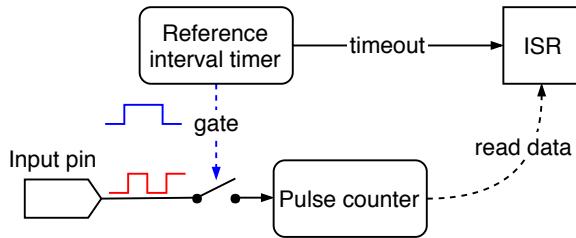
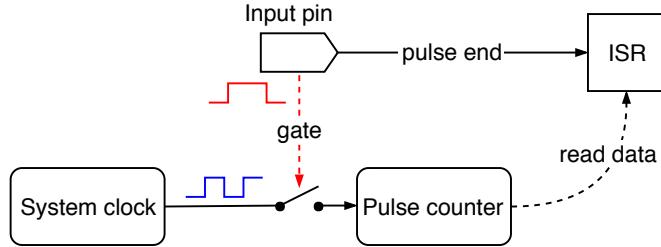
One timer generates the time window and its output gates the input of another, configured as a pulse counter. At the end of the measurement window an interrupt is generated and we can read the pulse count from the counter's register.

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

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

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

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

**Figure 8.1:** Direct frequency measurement method**Figure 8.2:** Reciprocal frequency measurement method

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 prescaler¹, 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.

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

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

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

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

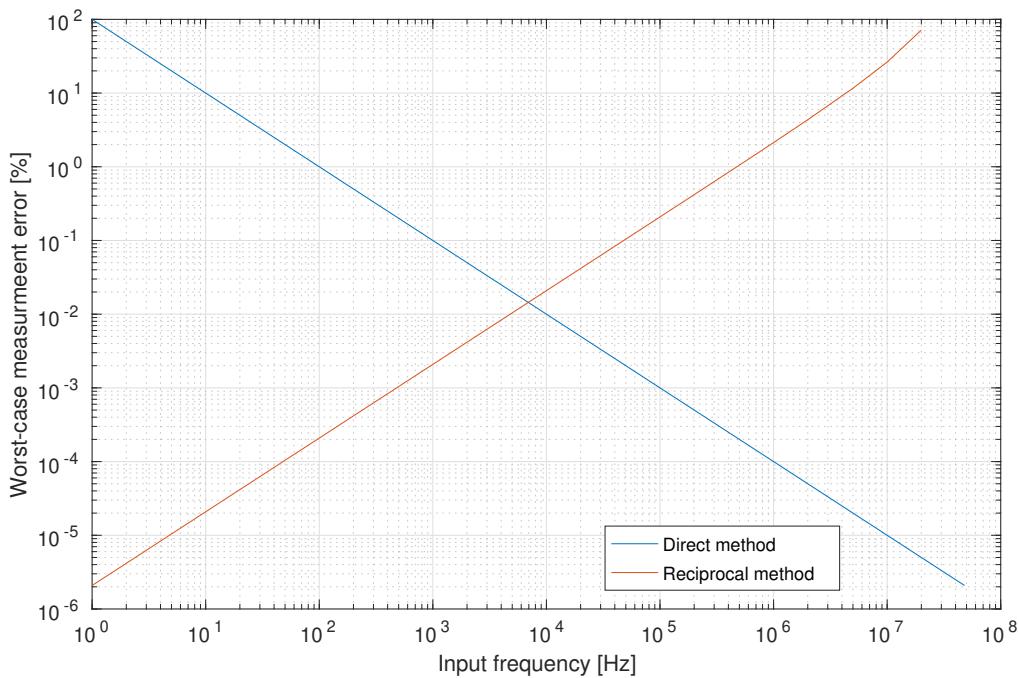


Figure 8.3: Worst-case error using the two frequency measurement methods with an ideal 48 MHz timer clock. The crossing lies at 7 kHz with an error of 0.015 %, or 1.05 Hz.

8.2 Waveform Generation

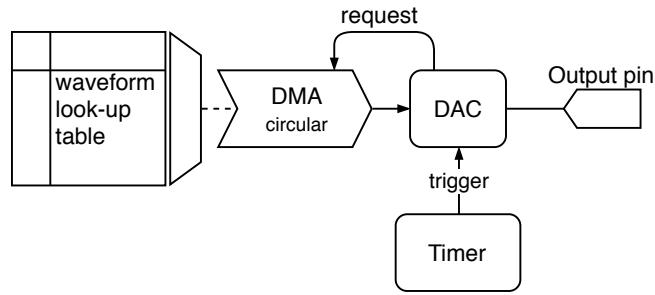
A waveform generator is a useful tool in many experiments and measurements. A sine stimulus is the basis of a lock-in amplifier; it can be used to measure impedance; with a frequency sweep, we can obtain the frequency response of an analog filter, etc. We can, of course, generate other waveforms, such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

8.2.1 Waveform Generation with DMA and a Timer

A straightforward implementation of the waveform generator is illustrated in figure 8.4. This approach has its advantages: it's simple and works entirely in the background, with no interrupt handling required. It could even be implemented entirely in software, using a loop periodically updating the DAC values, of course such approach is less flexible and we would run into problems with asynchronous interrupts.

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

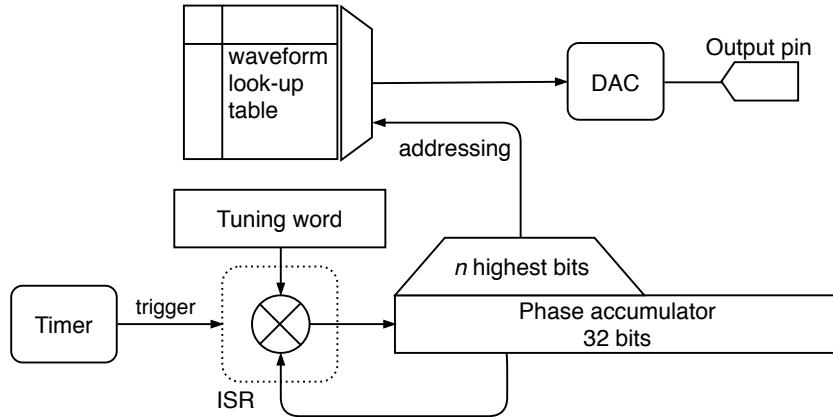
**Figure 8.4:** A simple implementation of the waveform generator

when observed with an oscilloscope, producing harmonic components similar to those of a square wave.

A major disadvantage of this simple generation method is given by the limitations of the used timer, which defines the output frequency. Its output trigger fires when the internal counter reaches a pre-defined value, after which the counting register is reset. The counting speed is derived from the system clock frequency f_c using a 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.

8.2.2 Direct Digital Synthesis

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

**Figure 8.5:** A block diagram of a direct digital synthesis waveform generator

A diagram of a possible DDS implementation in the STM32 firmware is shown in figure 8.5. It is based on a *numerically controlled oscillator* (NCO). NCO consists of a *phase accumulator* register and a *tuning word* which is periodically added to it at a constant rate in a timer interrupt handler. The value of the tuning word determines the output waveform

frequency. The look-up table must have a power-of-two length so that it can be addressed by the n most significant bits of the phase accumulator. An additional control word could be added to this address to implement a phase offset for applications like a phase-shift modulation.

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

DDS Implemented in Hardware

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

screenshots of a real demo, also reference to all-about-direct-digital-synthesis.pdf

Part III

Firmware Implementation

Chapter 9

Application Structure

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

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

references

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

9.1 User's View of GEX

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

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

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

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

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

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

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

■ 9.2 Functions of the Core Framework

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

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

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

9.3 Resource Allocation

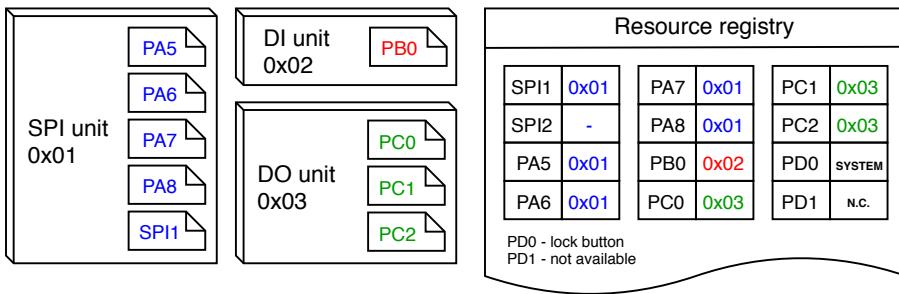


Figure 9.1: An example allocation in the resource registry

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

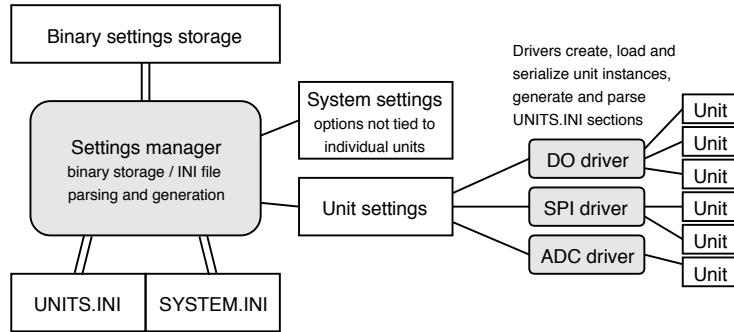
To prevent a multiple access, the firmware includes a *resource registry* (fig. 9.1). Each individual resource is represented by a field in a resource table together with its owner's callsign. Initially all resources are free, except for those not available on the particular platform (i.e. a GPIO pin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

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

9.4 Settings Storage

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

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

**Figure 9.2:** Structure of the settings subsystem

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

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

add a sample large INI file as an attachment

9.5 Functional Blocks

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

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

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

The unit driver handles commands sent from the host PC, initializes and de-initializes the unit based on its settings, and implements other aspects of the unit's function, such as

periodic updates and interrupt handling. Unit drivers may expose public API functions to make it possible to control the unit from a different driver, allowing the creation of "macro units".

9.6 Source Code Layout

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

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

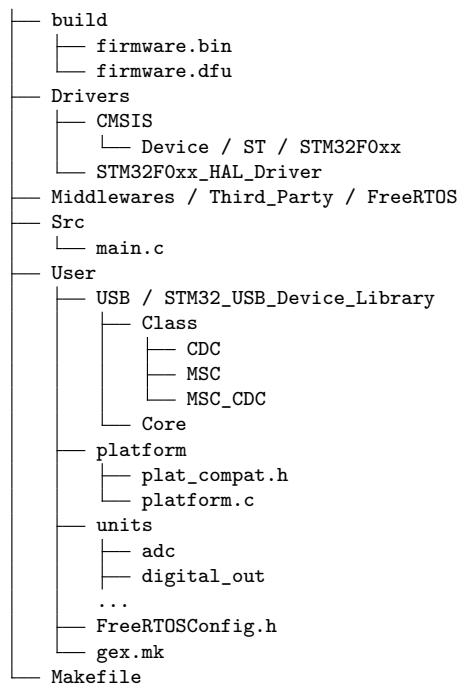


Figure 9.3: The general structure of the source code repository

9.7 Communication Ports

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

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

■ 9.7.1 USB Connection

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

■ 9.7.2 Communication UART

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

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

■ 9.7.3 Wireless Connection

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

Link to
tinyframe
description

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

■ 9.8 Message Passing

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

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

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

9.9 Interrupt Routing

Interrupts are an important part of almost any embedded application. They provide a way to rapidly react to asynchronous external or internal events, temporarily leaving the main program, jumping to an interrupt handler routine, and then returning back after the event is handled. Interrupts are also the way FreeRTOS implements multitasking without a multi-core processor.

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

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

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

Chapter 10

Communication Protocol

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

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

GEX uses a framing library *TinyFrame*, developed likewise by the author, but kept as a separate project for easier re-use in different applications. The library implements frame building and parsing, checksum calculation and a system of message listeners. An interested reader may find more technical details and the API in its documentation.

10.1 Frame Structure

Message frames have the following structure:

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

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

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

10.2 Message Listeners

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

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

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

10.3 Designated Frame Types in GEX

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

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

10.4 Bulk Read and Write Transactions

The bulk read and write transactions are generic, multi-message exchanges which are used to transfer the INI configuration files. They could possibly 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, owing to the numerous explanatory comments, which would require a prohibitively large buffer. Further, the GEX module may need some time to process a part of the message before it can receive more data, which is easily achieved by this multi-part transport where each chunk must be confirmed before proceeding to the next.

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 corrupt state. As hinted in the introduction of this chapter, a transaction is defined by sharing a common frame ID. Thus, all frames in a bulk transaction must have the same ID, otherwise the ID listeners won't be called and the transaction will fail.

10.4.1 Bulk Read

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

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

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

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

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

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

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

■ 10.4.2 Bulk Write

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

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

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

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

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

■ 10.4.3 Persisting the Changed Configuration to Flash

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

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

■ 10.5 Reading a List of Units

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

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

10.6 Unit Requests and Reports

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

10.6.1 Unit Requests

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

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

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

10.6.2 Unit Reports

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

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


Chapter 11

Wireless Interface

Four methods of a wireless connection have been considered: Bluetooth (e.g. with CC2541), WiFi with ESP8266, LoRA or GFSK-modulated 868 MHz link with SX1276, and a 2.4 GHz link with NRF24L01+. Bluetooth was dismissed early for its complexity and ESP8266 for its high consumption in continuous reception mode, although both solutions might be viable for certain applications and with more time for evaluation. The SX1276 and NRF24L01+ have both been tested using the first GEX prototype, confirming its usefulness as a hardware development tool.

11.1 Comparing SX1276 vs. NRF24L01+

Part IV

Hardware Design

Appendices

