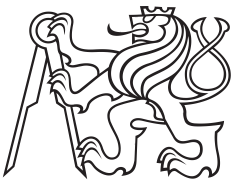
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Measurement

Learning and automation GPIO platform

Ondřej Hruška

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

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

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

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

Department / Institute: **Department of Measurement**

Study program: **Cybernetics and Robotics**

Branch of study: **Sensors and Instrumentation**

II. Master's thesis details

Master's thesis title in English:

Learning and Automation GPIO Platform

Master's thesis title in Czech:

Výuková a automatizační GPIO platforma

Guidelines:

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

Bibliography / sources:

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

Name and workplace of master's thesis supervisor:

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

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

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

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

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

III. Assignment receipt

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

Date of assignment receipt Student's signature

Declaration

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

V Praze, 27. května 2018

.....

Acknowledgements

blabla

Abstract

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

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

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

Keywords:

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

Abstrakt

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

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

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

Klíčová slova:

Překlad názvu: Výuková a automatizační GPIO platforma

Contents

Figures

Part I

Introduction

Chapter 1

Motivation

Prototyping, design evaluation and the measurement of physical properties in experiments make a daily occurrence in the engineering praxis. Those tasks 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.

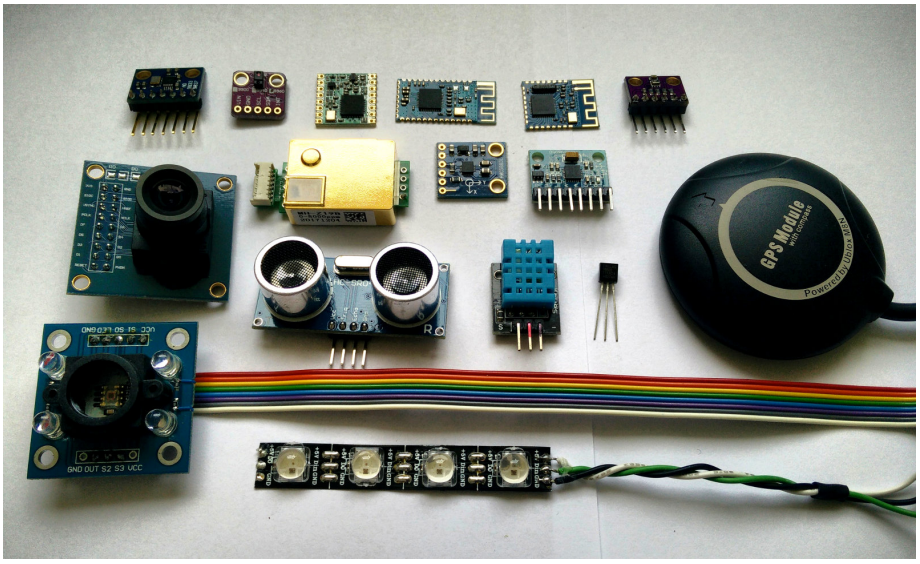


Figure 1.1: A collection of intelligent sensors and devices, most on breadboard adapters: (from top left) a waveform generator, a gesture detector, a LoRa and two Bluetooth modules, an air quality and pressure sensor, a CO_2 sensor, a digital compass, an accelerometer, a GPS module, a camera, an ultrasonic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

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

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

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

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

1.1 The Project's Expected Outcome

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

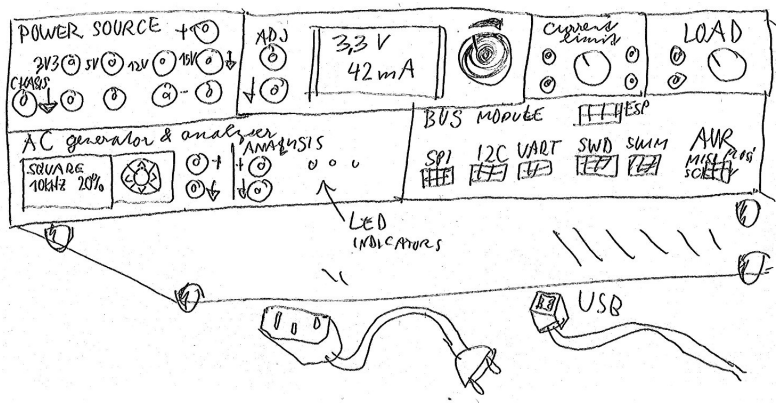


Figure 1.2: An early (2016) sketch of a universal bench device including a power supply, electronic load, a signal generator and a bus module. The bottom half of the panel is in a large part implemented by GEX.

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

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

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

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

Chapter 2

Requirement Analysis

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

2.0.1 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 records signals varying in time.

Certain tasks, such as capturing transient effects on a thermocouple when inserted into a flame (an example from developing fire-proof materials) demand level triggering similar

to that of oscilloscopes. The converter continuously measures the input voltage and a timed capture starts only after a set threshold is exceeded. This can be accompanied by a pre-trigger feature where the timed capture is continuously running and the last sample is always compared with the threshold, recording a portion of the historic records together with the following samples.

2.0.3 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, I^2C, UART/USART
  - Dallas 1-Wire
  - NeoPixel (addressable LED strips)
- **Communication with the host computer**
  - USB connection as virtual serial port or direct endpoint access

- Connection using plain UART
- Wireless attachment

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

2.3 Microcontroller Selection

The STM32F072 microcontroller was chosen for the built prototypes and the initial firmware, owning 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, I^2C, and USART peripherals, among others. It supports crystal-less USB, using the USB SOF packet for synchronization of the internal 48 MHz RC oscillator; naturally, a real crystal resonator will provide better timing accuracy.

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

2.4 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 insertion 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

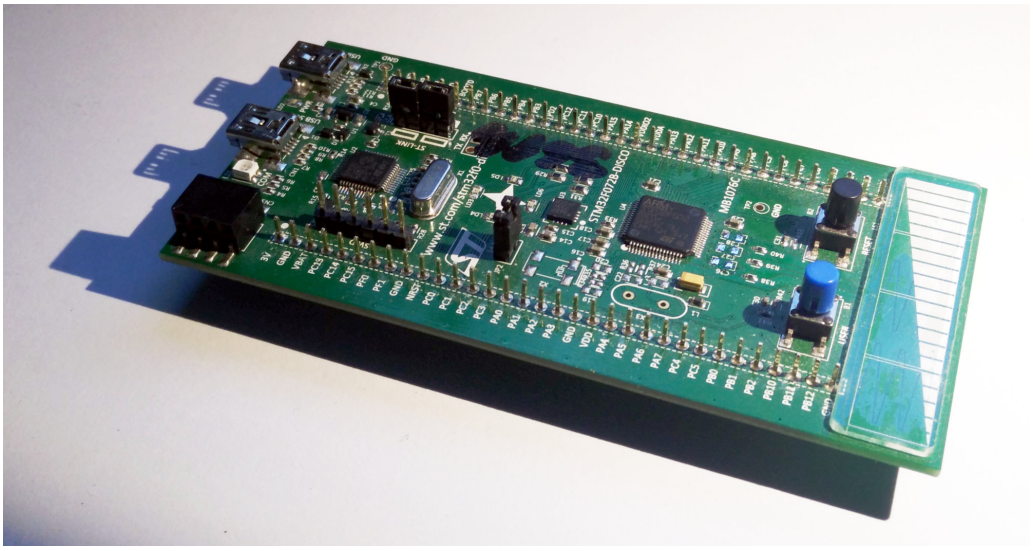


Figure 2.1: A Discovery board with STM32F072 that can be used to run the GEX firmware

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.

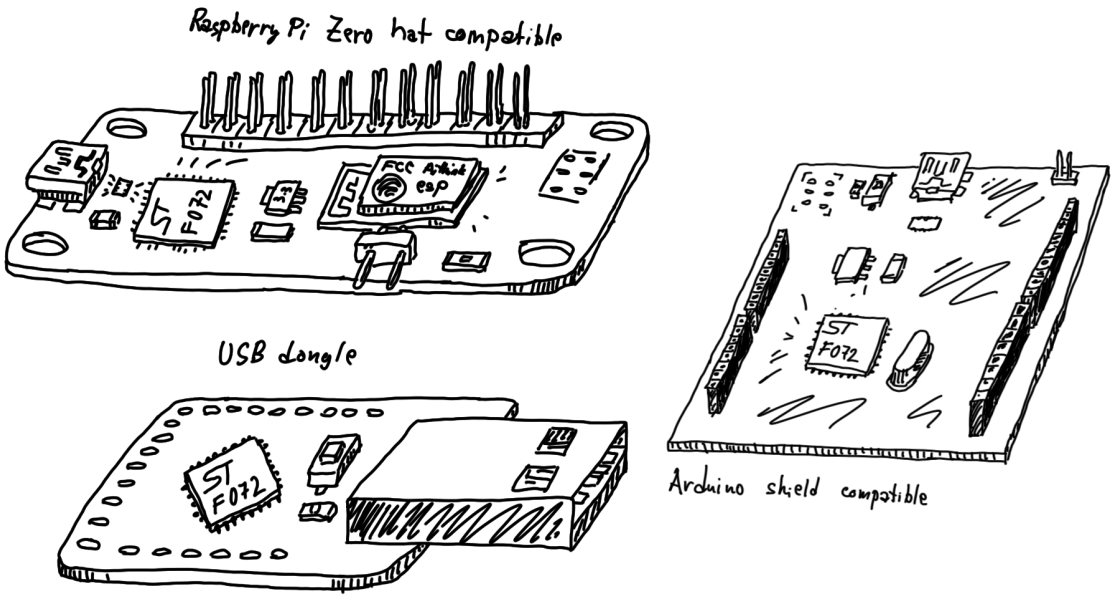


Figure 2.2: A sketch of three possible form factors for the GEX hardware prototype. Note the ESP8266 module which was considered as an option for wireless access but was eventually not used due to it's high current usage, unsuitable for battery operation.

Chapter 3

Existing Solutions

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

3.1 Bus Pirate

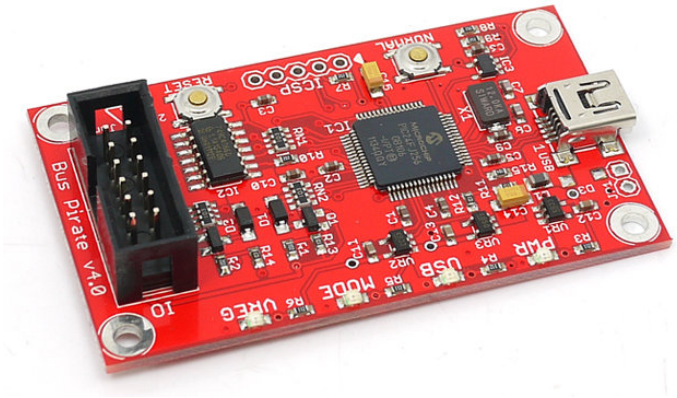


Figure 3.1: Bus Pirate v.4 (picture by *Seeed Studio*)

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, I^2C, 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

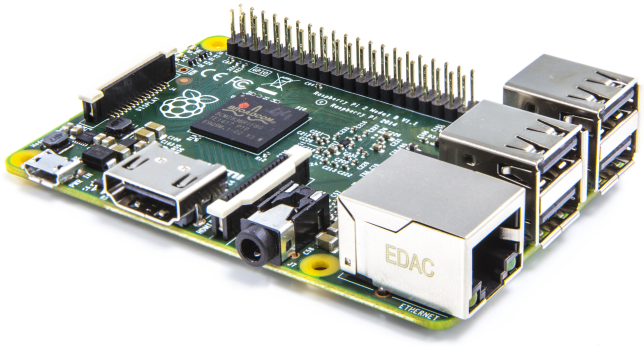


Figure 3.2: Raspberry Pi 2 (picture by *Raspberry Pi Foundation*)

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 IØ, 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

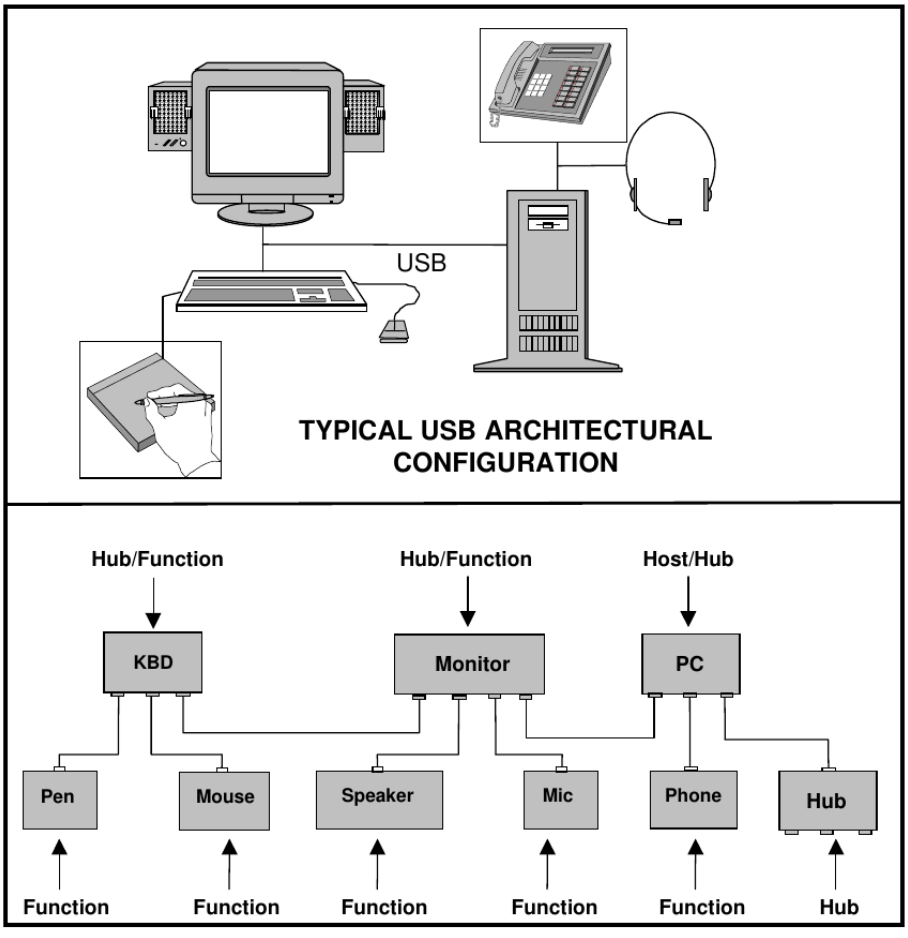


Figure 4.1: A diagram from the USB specification rev. 1.1 showing the hierarchical structure of the USB bus; The PC (Host) controls the bus and initiates all transactions.

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.

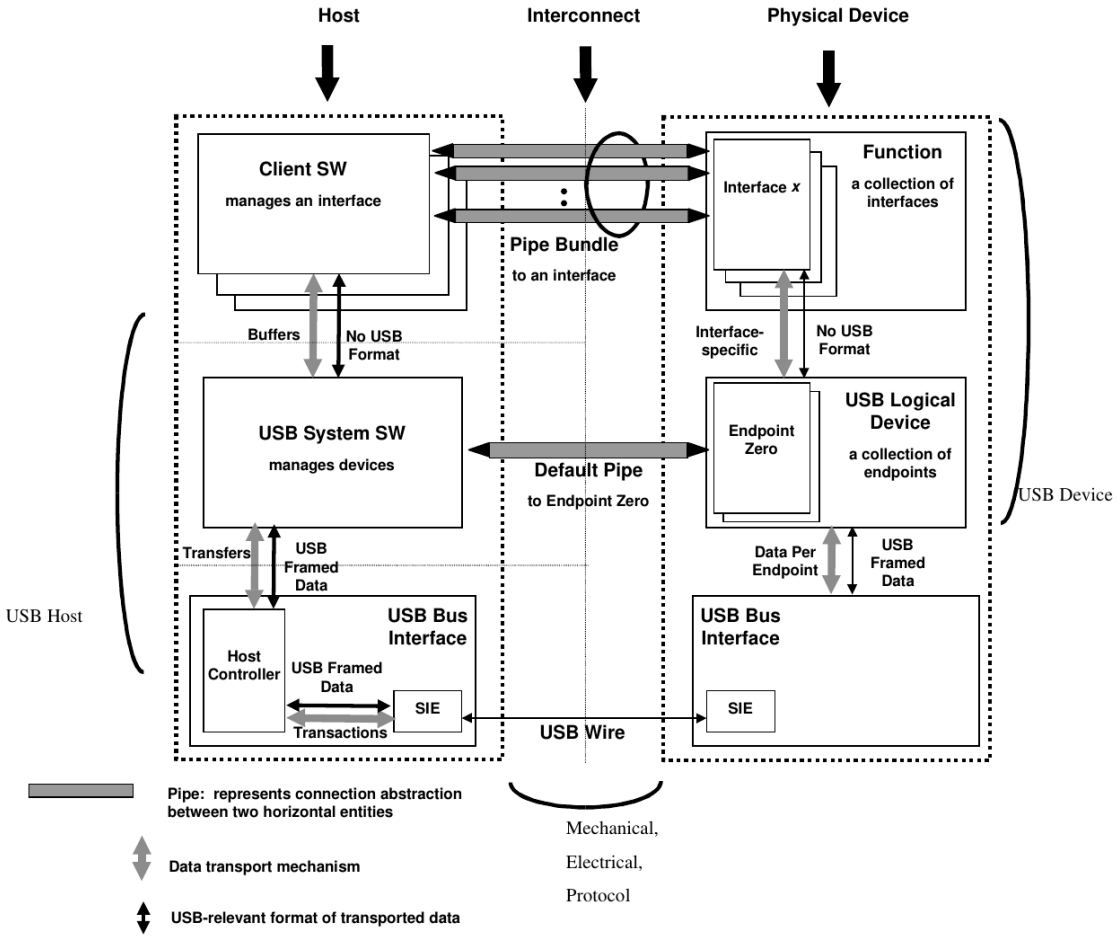


Figure 4.2: A detailed view of the host-device connection (*USB specification rev. 1.1*)

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

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

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

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

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

add reference to the document

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

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

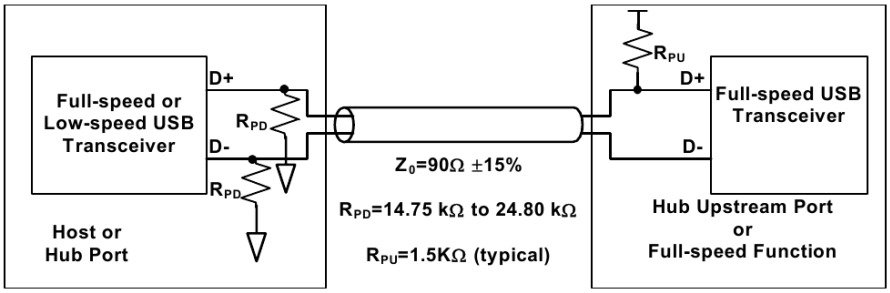


Figure 4.5: Pull-up and pull-down resistors of a Full Speed function, as prescribed by the USB specification rev. 2.0

4.2 USB Physical Layer

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

The USB cable contains 4 conductors:

- V_{BUS} (+5 V)
- D+
- D–
- Ground

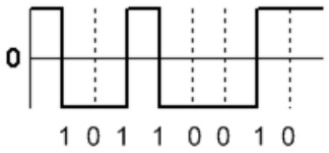


Figure 4.4: NRZI encoding example

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

USB revisions are, where possible, backwards compatible, often even keeping the same connector shape. The bus speed is negotiated by the device using a 1.5 kΩ pull-up resistor to 3.3 V on one of the data lines: for Full Speed, D+ is pulled high (fig.), 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.

4.3.1 Mass Storage Class

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

references

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

links

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

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

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

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.

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

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 can be considered a predecessor of USB in some aspects. RS-232 was once a common way of connecting modems, printers, mice and other devices to personal computers. UART framing is also used in the industrial bus RS-485.

UART, as implemented by microcontrollers, is a two-wire full duplex interface that uses 3.3 V or 5 V logic levels. The data lines are high when idle. A frame starts by a start-bit (low) followed by n data bits (typically eight), an optional parity bit and 0.5 to 2 stop bits (high). Variants with fewer or more bits exist, especially in older hardware. The parity bit can be odd, even, or missing entirely. A stop bit is usually 1 clock cycle long; other lengths are used in protocols derived from UART, such as in the SmartCard interface. reference

figure

UART and USART are two variants of the same interface. USART includes a clock signal and should therefore support higher frequencies. UART timing relies on a well-known clock speed and is synchronized by start bits. In RS-232 the two data lines (Rx and Tx) are accompanied by RTS (Ready To Send), CTS (Clear To Send) and DTR (Data Terminal Ready) that facilitate handshaking and hardware flow control.

examples

7.2 SPI

SPI (Serial Peripheral Interface) is a point-to-point or multi-drop master-slave interface based on shift registers. 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 other wires 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 at the same time. A bus master asserts the SS pin of a slave it wishes to address and then sends data on the MOSI line while receiving a response on MISO. The slave doesn't know the command before the first byte is completed, so it usually responds with zeros or sends a status byte in this phase.

figure

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.

examples

7.3 I2C

Last of the three common protocols covered here is be I2C. It's a two-wire, open-drain bus that supports multi-master operation. I2C is more complicated than either UART or SPI; it supports 3 speeds .

speeds

diagrams and more from the specification

7.4 1-Wire

1-Wire uses a bi-directional data line that can also power the connected devices, giving the bus its name. TODO „„,

7.5 NeoPixel

...m„,

Part III

Firmware Implementation

Chapter 8

Application Structure

GEX was designed to be modular and easy to extend. At its core lies a general framework that provides services to the functional blocks configured and used by the user script running on the host PC. Functional blocks, or internally called *units*, implement functions like SPI, I2C, GPIO access etc.

In this chapter we'll focus on the general function of the GEX module, look at implementation details of the core framework, and in the next chapter some space will be given to each of the functional blocks.

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. All design choices will nevertheless be explained as well as possible.

8.1 User's View of GEX

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

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

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

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

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

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

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

8.2 The Core Framework Functions

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:

- Hardware resource allocation
- Settings storage and loading
- Functional block initialization
- Communication port with different back-ends (USB, UART, wireless)
- Message sending and delivery
- Interrupt management and routing to functional blocks
- Virtual mass storage to allow editing of the configuration files

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.

8.2.1 Source Code Layout

Looking at the source code repository, at the root we'll find device specific driver libraries and files provided by ST Microelectronics, the FreeRTOS middleware, and a folder `User` containing the GEX firmware. This folder is a git submodule. The GEX core framework consists of everything in `User`, excluding the `units` folder. The USB Device library, which had to be modified to support a composite class, is stored inside the `User` folder as well. Hardware configuration, such as the status LED position or clock settings, are defined using compile flags set in the top level Makefile.

8.3 Functional Blocks

GEX's functional blocks, internally called *units*, have been mentioned a few times but until now haven't been properly explained. GEX's user-facing functions are implemented in *unit drivers*. Those are stand-alone modules that the user can enable and configure using the configuration file. In principle, there can be multiple instances of each unit type. However, in practice, we have to work with hardware constraints: there is only one ADC peripheral, two SPI ports and so on. This limitation is handled using resource allocation, as explained below.

Each unit is defined by a section in the configuration file `UNITS.INI`. It is given a name and a *callsign*, a number which serves as an address for messages from the host PC, or, conversely, to indicate which unit sent an event report (such as a pin change interrupt). A unit is internally represented by an object that holds its configuration, internal state, and a link to the unit driver. The driver handles commands sent from the host PC, initializes and de-initializes the unit based on its settings and implements other aspects of a unit's function, such as periodic updates and interrupt handling.

8.4 Resource Allocation

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

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

used by the core framework are taken by a virtual unit `SYSTEM` to prevent interference by user's units. This is the case of the status LED, the LOCK button, GPIO pins used for connecting USB, communication UART, the wireless module and the crystal oscillator, as well as the timer/counter which keeps the system timebase.

resources diagram

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

The mentioned INI files that 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.

a diagram

8.6 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.

8.6.1 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.

8.6.2 Communication UART

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

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

8.6.3 Wireless Connection

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

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

LINK

8.7 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 XX .

Link to tinyframe description

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 8.3), 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.

Part IV

Hardware Design

Appendices