Evaluating the usefulness of USB for real-time robotics applications

Mihai Pomarlan

Supervisors: Dr. Alin Albu-Schäffer

Prof. Dr. Ing. Ivan Bogdanov

Catedra "Roboți Industriali"
Universitatea Politehnica Timişoara

Acknowledgements

I would like to take this opportunity to thank several people who have been of great assistance during the work on this thesis. I thank my supervisors, Prof.-Dr. Ivan Bogdanov and Dr. Alin Albu-Schäffer, who made this thesis possible in the first place. To Mr. Klaus Jöhl I owe many thanks for all the help he has provided throughout the thesis. I would also like to thank Mr. Reintsema Detleff for showing me an introduction to USB, Mr. Stefan von Dombrovsky for all the support he provided on the Linux system, and Mr. Carsten Preusche for providing the resources to do experiments with.

Table of Contents

1 Motivation 6
2 The USB communication protocol 7
2.1 Features overview 7
2.2 Establishing a connection 7
2.2.1 Enumeration and descriptors 7
2.2.2 Device classes 9
2.2.3 Bandwidth negotiation 9
2.3 Communication pipes 9
2.3.1 Endpoints and pipes 9
2.3.2 Transfer types 10
2.3.3 Transfer acknowledgment 11
2.3.4 Usage of frames 11
3 The USB device firmware (USBP) 12
3.1 Firmware structure 12
3.1.1 File structure 13
3.1.2 Protocol handling 14
Limitations 16
Further developments 17
3.1.3 Hardware abstraction 19
Hardware bridge 20
Performance measurements 21
Hardware portation 22
3.1.4 Configuration 23
Callback writing 24
Using the SOF and Tx callbacks for data transfers 25
System configuration and integration 26
3.1.5 Classes 27
4 The netX-based ROKVISS joystick 29
4.1 The netX chip 29
4.1.1 Features overview 29
4.1.2 EtherCAT 29
4.2 The control board 30
4.3 The joystick software 31
4.3.1 Functionality 31
The SPI “drivelet” 31
The system timer 32
The USB firmware: USBP 32
The joystick software 33
4.4 Performance of firmware 36
4.5 The joystick demo application 37
5 The USB device drivers 40
5.1 USB communication overview 40
5.2 Windows XP – the HID driver 42
5.2.1 HID class: overview 43
5.2.2 Using the HID driver: usbhidio 44

5.2.3 Running a USB communication in Matlab (Windows XP)..... 44
5.2.4 Controlling position from MATLAB via USB..... 47
5.3 Linux – libusb..... 50
5.3.1 Running a USB communication in Matlab (Linux)..... 51
5.3.2 Running a USB communication in joystick demo..... 52
5.4 QNX – usbd..... 54
5.4.1 Running a USB communication on QNX..... 55
6 Conclusions..... 62
7 Annexes..... 64
7.1 A1: CD contents..... 64
7.2 A2: Joystick program..... 64
7.3 A3: Monitoring a motor with the netX..... 65
7.4 A4: Flasher App..... 65
7.5 A5: Testing USB on Windows..... 66
7.5.1 A5.1: Joystick_test for Windows..... 66
7.5.2 A5.2: Motor position control in Simulink..... 66
A5.2.1: Tuning the PI controller..... 67
7.6 A6: Testing USB on Linux..... 67
7.6.1 A6.1: Joystick test for Linux..... 67
7.6.2 A6.2: Joystick demo Linux..... 67
7.7 A7 Testing USB on QNX..... 68

1 Motivation

Readily available in computers today, USB is the bus of choice for a wide range of peripherals, from keyboards to webcams. Its flexibility and compactness surpasses that of previous PC interfaces, like the serial and parallel ports. It offers plug-and-play capability, good data rates, and it is cheap and reliable.

It is also very common in microcontrollers; even low-range, 8-bit controller families have members with USB hardware on them. USB allows very compact circuit boards to be designed, as evidenced by the thriving FLASH-stick industry.

However, hard real-time applications are quite rare for USB. Special buses like SERCOS, Firewire or EtherCAT are more typically employed in such cases. These either require special, non-common hardware on the computer (SERCOS, Firewire) or the device side (EtherCAT).

It is therefore interesting to ascertain the usefulness of USB for hard real-time applications, precisely because a USB connection can be established and maintained with regular, easily available hardware.

During the first part of this diploma work, a USB peripheral was programmed: the new ROKVISS force-feedback joystick. This joystick has control electronics that allows either a USB or an EtherCAT connection to be used. At this point, the USB connection is operational. A USB firmware was developped for the joystick, however this firmware is designed to be easily portable both to new hardware platforms and to new projects. In addition to the joystick's usual functionality, timing has been implemented, in order to assess the USB communication's performance.

In the second part of the diploma work, the joystick was connected to PCs running different operating systems. As a measure of communication performance, round-trip time was monitored in each case. Round-trip is measured by the joystick, between the moment it prepares some data to be sent to the computer, and the moment it receives a reply to that particular packet of data.

The operating systems tested were: Windows XP, Linux (regular OSes) and QNX (real time Operating System). This permits the evaluation of USB communication both in usual, as well as special real-time setups, and therefore, deciding for what kinds of applications a USB connection is useful, for example motor control with a connection to Simulink.

2 The USB communication protocol

This chapter is based on the “Universal Series Bus Specification” document revision 2.0 and is a very succinct presentation of the USB standard. Its purpose is to ease understanding of the following chapters by introducing some key concepts about USB.

2.1 Features overview

Some of the more significant features of USB are the following:

- “polled” bus. There are one Host and at most 127 devices on the bus, and it is the Host that initiates any data transfer, whether toward or from the Host. All transfers have the Host as one of the end points[2.1, 2.2].
- is specified for low speed (1.5Mbps max baud rate), full speed (12Mbps max baud rate) and high speed (480Mbps max baud rate) connections[2.3].
- can provide power to the devices on the bus[2.3].
- differential data transfer[2.3].
- communication is split in frames, generated each millisecond for low/full speed devices and 125 microseconds for high speed[2.4].
- cheap and readily available on present-day computers and microcontrollers.

2.2 Establishing a connection

2.2.1 Enumeration and descriptors

USB is a fundamentally asymmetric bus, in the sense that a very clear distinction is made between a bus “Host”, which controls all activity on the bus, and the various USB devices connected to the bus that respond to the Host's requests. On a bus there must be only one Host[2.2].

When the Host detects that a new device has been connected, it issues a series of requests toward that device in order to identify it. The device will respond by sending back descriptors, which are binary sequences describing various aspects of the device [2.5, 2.6].

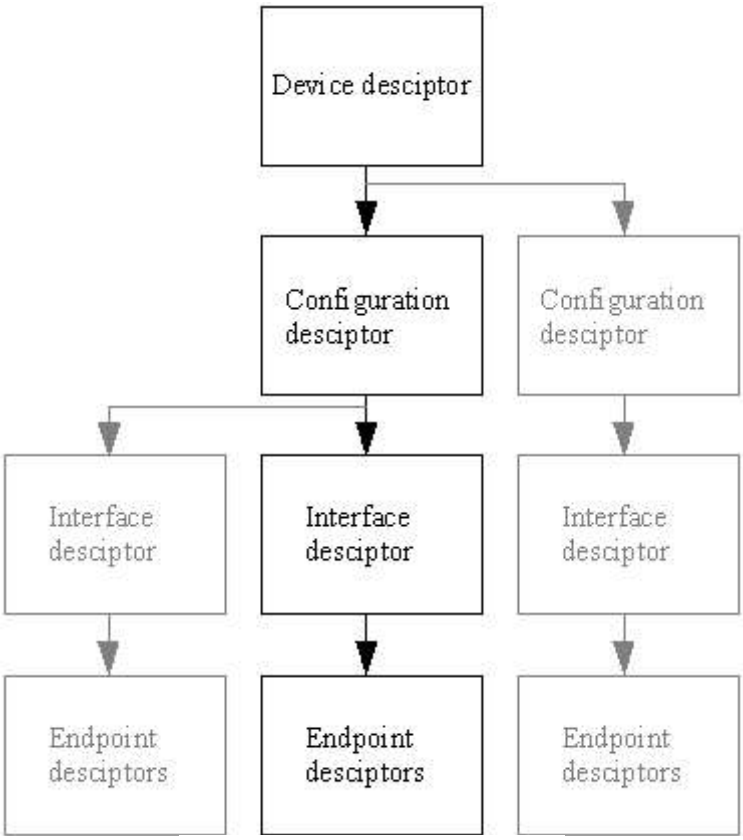

Fig 2.1 Descriptor hierarchy

A device has therefore a “device descriptor”, which gives information on the manufacturer of the device and the device type, at least one “configuration” descriptor that gives information on how much power the device needs, at least one “interface” descriptor giving information on what kind of connections the device wishes to form with the Host, each of these connections being associated to an endpoint descriptor. The descriptors form a kind of hierarchy, in that the device descriptor also provides the number of configuration descriptors, and each configuration descriptor provides the number of interface descriptors associated with it[2.6].

Other descriptors may be included depending on the type of device. Optionally, string descriptors may also be present, allowing a human user easy recognition of the device and its various functions[2.6].

Once identified, if accepted as correctly functioning, the device is assigned an address by the Host. A configuration and an interface are selected, and normal communication can begin[2.7].

Each of the operations mentioned above corresponds to a standard request: GET DESCRIPTOR, SET ADDRESS, SET CONFIGURATION, SET INTERFACE. All requests are issued by the Host to a device[2.7].

On the Host side, once a device is identified, the Host must search for a driver that will be able to communicate with that device. The search is based on the device's attributes, for example its Vendor and Product identifiers, or its class[2.5].

2.2.2 Device classes

Either in the device descriptor or in the interface descriptors, the USB device can signal its belonging to a certain device class. This provides some further information to the Host about what the device is capable of, what other features may need configuring, what driver can work with the device[2.8].

Each device or interface class is described by its own standard, which places various minimal requirements on what a device must be capable of doing and maximum capability bounds. An exception is a special class, the "Vendor-defined" class, on which no restriction is made[2.19, 2.20].

Examples of classes are Human Interface Device, which includes mice and keyboards, and Mass Storage Device, which includes FLASH sticks[2.19].

2.2.3 Bandwidth negotiation

There is a limited amount of bandwidth available on the USB, and all devices present on the bus must be able to share it. If a new device is connected, and the bandwidth it requires is more than what is not used by the already connected devices, then the Host will "refuse" to connect to it[2.9].

Sometimes this may be solved by the device providing several configurations, with different bandwidth requirements. If one of these requirements can still be accommodated on the bus, the Host can select it and connection is possible. Otherwise, the Host prioritizes the connections it already has[2.9].

2.3 Communication pipes

2.3.1 Endpoints and pipes

At the simplest level, a device is, to the Host, "a collection of independent endpoints". Each endpoint on a device will correspond, once the device is connected, to a single communication pipe between Host and device[2.10].

Every device has at least one endpoint, "endpoint 0", which is used to issue requests

at. This endpoint is bidirectional, because it can accept data from the Host (requests and parameters) as well as send data back (descriptors, for example)[2.10].

Unless configured as a "control endpoint", any other endpoint on a device is one-directional, either IN- it sends data toward the Host- or OUT- it receives data from the Host. At most, a device can have a total of 31 endpoints, including endpoint 0. Every control endpoint will actually require using two endpoints for the two directions of communication[2.10].

Each endpoint, including endpoint 0, has a maximum packet size that gives in bytes the largest data packet that can be received or sent[2.10].

Endpoints are "logical" entities, that is they do not correspond to some salient hardware feature like a different cable connection or a different chip (but they do have different data buffers). Software on the device is responsible with configuring the endpoints as appropriate, and keep the configuration consistent with what its descriptors say[2.10]. For example, the same chip can declare itself as having a modem-like interface with three nonzero endpoints and a speaker-like interface with a single non-zero endpoint. As long as it implements the interfaces consistently (switches interfaces exactly as the Host requests), correct communication is possible.

Each endpoint will have at most one data transfer during a USB frame on a full-speed bus, or at most three on a high-speed one[2.11].

In this document, "endpoint" and "[communication] pipe" will be used interchangeably.

2.3.2 Transfer types

Besides direction, an endpoint is characterized by the transfer type it uses. The USB standard distinguishes four types of transfers:

- control: used mainly by endpoint 0, it is the only transfer type with a data structure imposed by the standard. Requests and responses to requests are sent via control transfers. They are classified as "non-periodic" (rare) and can tolerate large latencies (the Host will not deliver them immediately if other non-bulk transfers are pending). Limited in transfer size to 8 bytes (low-speed), 16, 32 or 64 bytes (full and high speed)[2.12].
- bulk: "non-periodic", not time critical, not limited in size (except by available bandwidth) transfer. Typically used by printers and mass storage devices[2.13].

- interrupt: “periodic” (can happen as often as every frame), time critical (the Host will not postpone it), limited in size to 8 bytes (low-speed), 64 bytes (full speed) or 1024 bytes (high speed). Typically used by mice, keyboards or joysticks, may also be useful to process control applications[2.14].
- isochronous: “periodic”, time critical, not limited in size (except for available bandwidth). Does not guarantee delivery of data, because it does not use acknowledgements nor retries. Typical applications are cameras and speakers[2.15].

2.3.3 Transfer acknowledgment

All transfer types, with the exception of the isochronous one, are ended with an acknowledgments token. This can either be an ACK (acknowledgment) in case of a successful transfer, a NAK (not acknowledged) in case of a transient error like a bit error from noise on the bus, or a STALL in case of a persistent failure[2.16].

Endpoint 0 may issue STALLs to signal that a request is not supported, or that its parameters are incorrect[2.16].

2.3.4 Usage of frames

A USB frame begins with a special packet, the SOF: start-of-frame. The devices can track the appearance of the SOF on the bus and use it for various synchronization purposes. The standard requirement is that the SOF is to be generated at reliable intervals, which are 1 millisecond for full-speed buses and 125 microseconds for high-speed ones, with at most 0.05% error[2.17].

Any transfer will happen during the frame time, and any endpoint can only transfer once per frame (or up to 3 times on a high speed bus). Of the frame time, 90%, or on high speed buses 80%, is used for “periodic” transfers, the rest is for bulk and control transfers. In case periodic transfers (interrupt and isochronous) do not fill all the time allocated to them in a frame, the Host can expand the slot for bulk and control transfers for that frame[2.18].

3 The USB device firmware (USBP)

Each USB device contains a functional part responsible for monitoring events on the USB- whether a reset has occurred, or a request from the host is pending, for example- and notify the device application program of such events. When the application program on the device wishes to respond, it does so through this functional part: the USB Logical Device firmware[3.1]. Hierarchically, the USB firmware is situated above the hardware level, and below the application or operating system running on the system. It communicates directly with the hardware or through OS functions, if an OS is present and requires special protocols for hardware access.

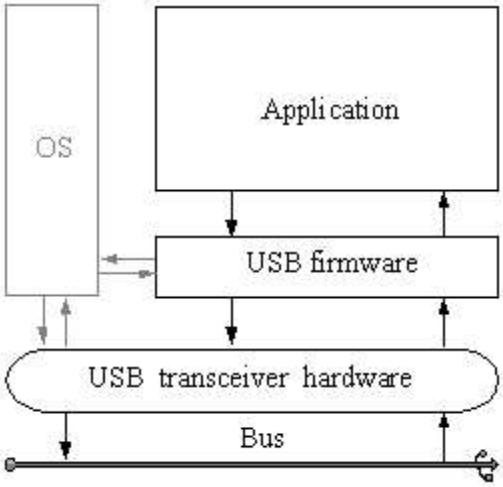

Fig 3.1 Software hierarcy on the device

In this chapter, the USB device firmware developed in the course of this diploma work (USBP) is presented. Its structure is shown and argumented for, descriptions of its features, functionality, and limitations are provided. Special emphasis is given to its portability between different projects and hardware platforms. USBP was made for the netX microcontroller because at the time there was no general USB firmware available for netX. However, it is applicable in embedded projects that use other microcontrollers than netX, by changing the USB hardware abstraction layer to fit the new platform. Chapter 4 shows USBP in use in a project and its performance.

3.1 Firmware structure

The firmware needs to implement, at least in part, the USB protocol specification for USB devices. It also needs to be aware of the project configuration (number and types of pipes, callbacks to the application etc) as well as monitoring and controlling the USB hardware.

However, the hardware part is dependent on the platform. The configuration is dependent on the project. The protocol handling part is, by definition, constant from project to project and platform, with the exception of the USB device class. Several device classes are specified by the USB standard[2.19], and it is very uncommon to have a device that needs all of them. Further, the standard allows a “Vendor-defined”

class, on which the standard makes no restrictions[2.20].

In light of this, the firmware is split into several parts that each handle a specific facet of the firmware's functionality: protocol, configuration, class, hardware.

- protocol part: as well as handling the standard requests as defined by the USB documentation, this part is responsible with providing most of the interface to the application or operating system. It defines the USB initialization and USB main functions.
- hardware part: serves as the USB hardware abstraction layer, enables the protocol part to access the status of the pipes, control their function, access the data. Optionally, provides performance measurement functions.
- configuration part: stores callback pointers, descriptors, pipe settings, declares compile switches.
- class part: implements code for class specific requests, is called by the protocol part when such requests are issued.

3.1.1 File structure

The diagram below shows an overview of the file structure of the USB device firmware.

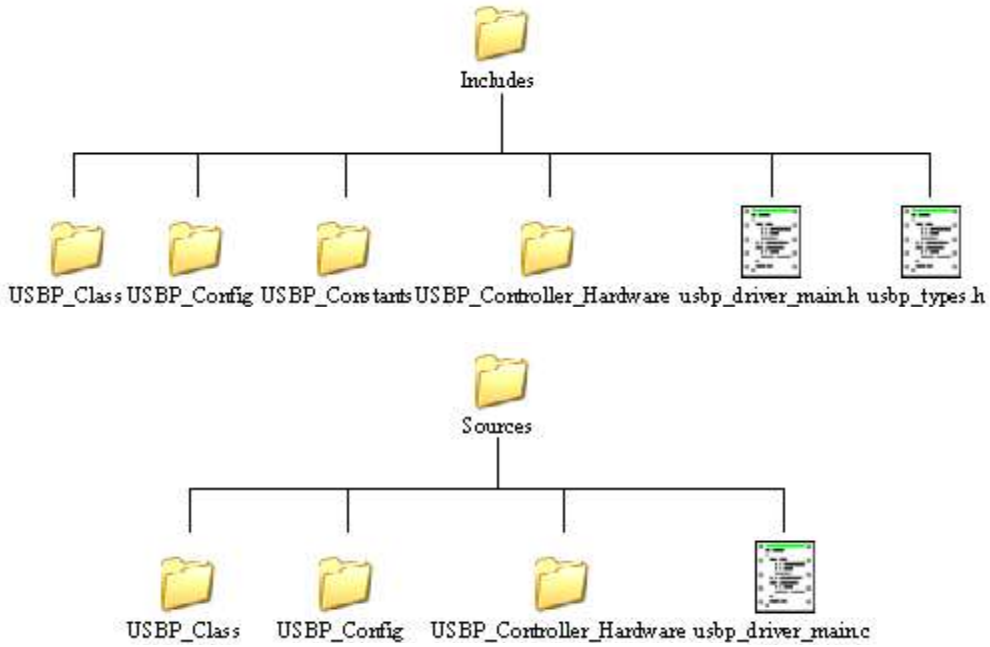

Fig 3.2 USBP file structure - overview

Each part of the firmware is given directories for their specific headers and source files respectively. Each part also has a central header, and usually it is this header that is included rather than every header individually. This allows for some changes to the file structure to happen without changing the include directives in other files.

The files usbp_driver_main.h and usbp_types.h are the files that the application or upper layer must include in order to use USBP.

The following subsections contain a more detailed description of each of the parts and their files.

3.1.2 Protocol handling

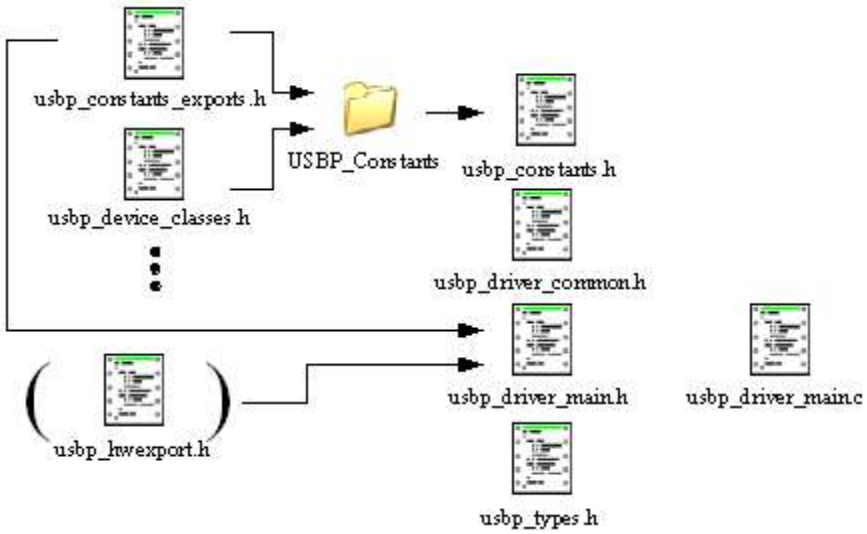

Fig 3.3 File structure detail: protocol handling

The USBP_Constants folder contains a set of headers declaring various USB standard-defined values, for example descriptor types, request identifiers, device and interface class identifiers, country values etc. One of these files, usbp_constants_exports.h, contains constants that are visible in the application level. This is so in order to enable the application to parse request headers, if it wishes to monitor (via callbacks) some of the USB requests, as well as reactivate pipes for input/output.

usbp_driver_common.h contains a definition for an internal structure used throughout USBP to store data about a request.

usbp_types.h is the basic type definition for USBP. It defines integer types according to their size: for example uint16 is unsigned integer on 16 bits, vint8 is a volatile signed 8-bit integer. It also defines a callback type for communication events (transmission or reception). This file might need to be changed on some platforms to preserve the definitions of the integers consistent with their names, e.g. ensure that int16 actually has 16 bits.

usbp_driver_main.h is the file that includes all exports of USBP toward the

application. In addition to the functions USBP_vInit and USBP_vMain_Function, it includes exports from the constants folder as well as exports from the hardware abstraction layer. See the sub chapter on USBP's hardware part for more info about these exports.

In order to use USBP, the application only needs to include (in this order) usbp_types.h and usbp_driver_main.h. It must ensure that USBP_vInit() is called before the USB is used. Afterwards, events on the USB are handled by USBP_vMain_Function(), which can be called either from a USB interrupt vector, or in a periodic task (polling).

The next figure illustrates a run of USBP_vMain_Function. It first checks for several events on the bus (like reset and start of frame). Optionally, it can notify the application layer of these events.

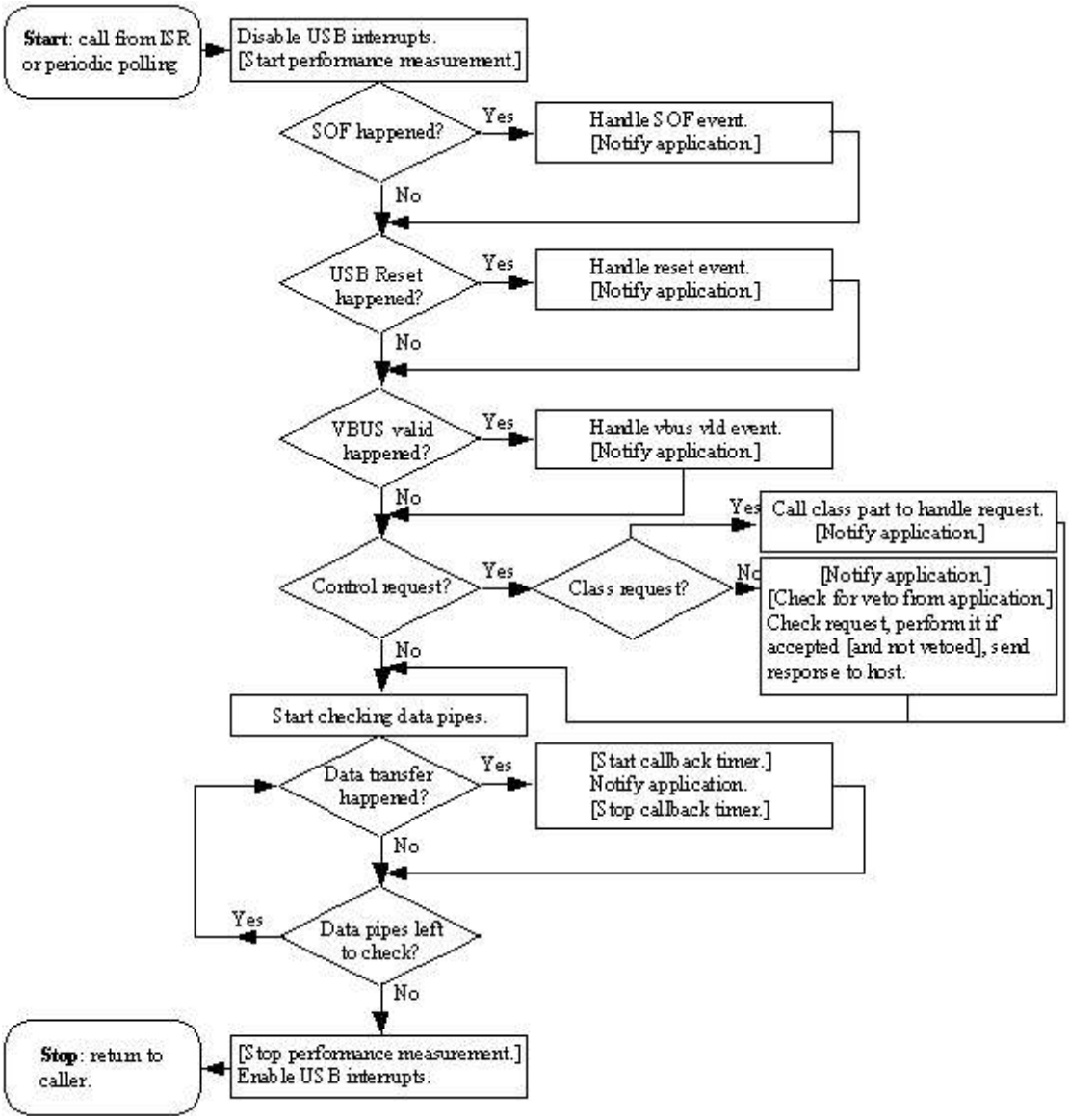

Fig 3.4 USBP_vMain_Function: logical diagram

Then it will check for events on pipe 0, the control pipe. If events occurred on this pipe, it means a request has been issued and must be handled by the appropriate functions. Optionally, the application is notified of some of the requests, and may have the possibility to "veto" them (declare that a particular request will not be serviced). In case that a request passes without veto, and is not found erroneous by USBP's sanity checking, it is performed. A response is sent to the USB Host to indicate the request's success or failure.

After finishing pipe 0, every data pipe is checked in turn for communication events. The application is notified when such an event occurs. Unlike the ones before, these notifications are not optional.

Performance measurement (described in its own subchapter in "3.1.3: Hardware Abstraction") may be done during USBP_vMain_Function.

Note that the USB interrupt is kept disabled for the entire run of USBP_vMain_Function. Only one instance of this function should run at a time.

Limitations

I modeled the firmware after the HID (human interface device) class, and this is so far the only class code that is provided. Any request that HID is required to support is supported by the firmware; requests unsupported by HID are not supported by the firmware either.

However, not all the limitations of HID apply. The firmware can be operated with High-Speed USB as long as a special handshake, the PING protocol employed for Bulk transfers, is not used. Any number and combination of pipes can be used, as opposed to one IN and at most one OUT as in the case of HID.

Although the class code provided is for an HID, it is possible to specify the device as a "vendor-class". I used this approach for connecting on Linux and QNX systems, because the HID driver on these systems is poorly documented.

Bulk and Isochronous transfers have not yet been tested, however there is nothing in the structure of the firmware that categorically rejects their use on full speed buses. On high speed buses, the PING protocol will have to be added for Bulk endpoints.

Only one instance of USBP_vMain_Function may run at any one time. Once started,

USBP_vMain_Function disables the microcontroler's USB interrupt (if interrupt use is enabled by compile switches), and keeps it disabled for the entire duration of its run, including application callbacks. Performance measurement functions should be used to determine if the time spent by USBP_vMain_Function and callbacks is long enough to threaten system responsiveness to USB.

Finally, access to the control, status, and data buffer bits of the USB hardware must be synchronous; the USB operations themselves can be asynchronous. This means, for example, that when the operation to enable a “Tx Ready” bit returns, then the bit is indeed set even if the transmission itself did not yet occur. This is not normally a problem on controllers with on-chip USB hardware, but it is an issue to keep in mind on systems that use an external USB chip.

Further developments

A few directions of improvement stand out:

- adding high speed bus protocol handling (PING for Bulk transfers)
- adding optimization options- for example, some of the currently supported requests should be made removable via compile switches, callback arrays do not need a callback for index 0, a single array is sufficient for both transmission and reception callbacks etc.
- adding support for several more requests (for example, set descriptor). Each new request should be added accompanied by a compile switch that disables it and re-enables the current “request not supported” code, so that unnecessary request handling code is not compiled.
- testing bulk and isochronous transfers.
- smoother support of multi-class devices.

Turning this package into a USB host firmware however is not recommended. While a Host firmware might benefit from a design split into protocol, class, configuration, hardware parts, the functionality of a Host is very different and requires different code.

There is no recipe for changing the protocol handling part of the USB firmware package, but some guidelines are possible to enumerate.

- the file structure should stay close to the present one. In particular, the split between protocol, class, configuration and hardware should be kept.
- whenever possible, use the firmware file and include structure to try to minimize the amount of change that propagates from file to file.

- additional code should come with compile switches that disable it.
- keep the Hardware Portation and System Configuration and Integration guidelines up to date.

3.1.3 Hardware abstraction

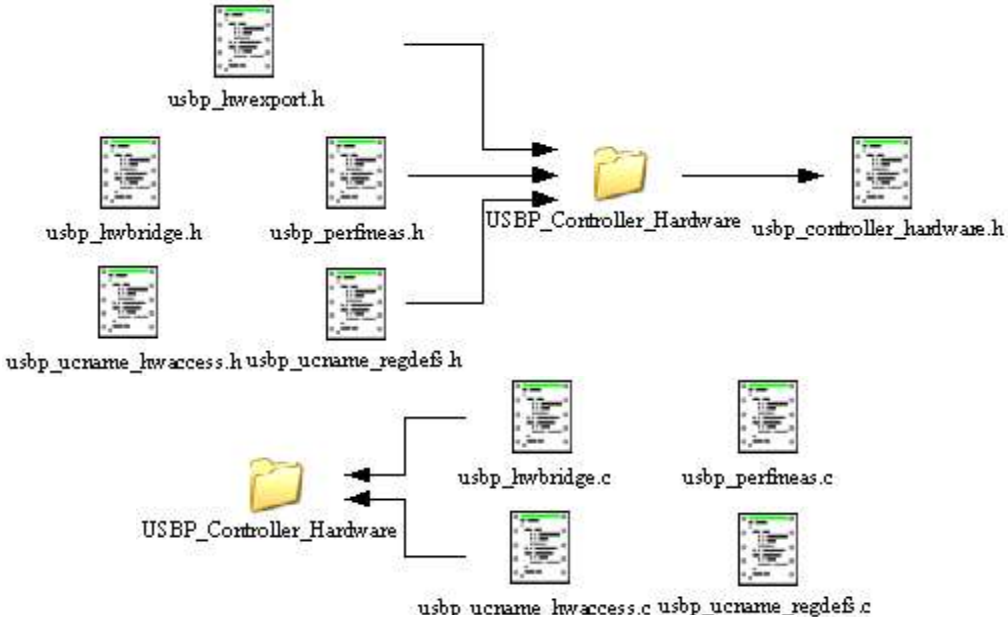

Fig 3.5 File structure detail: hardware access

usbp_controller_hardware.h includes all the hardware abstraction headers. Other files from USBP include this file, which allows the hardware abstraction headers to change names.

usbp_ucname_hwaccess.h and usbp_ucname_regdefs.h (where ucname is replaced with the name of the hardware platform) contain the hardware access and register declarations respectively.

usbp_hwbridge.h contains declarations for accessing the USB in a special situation- when the USB hardware is located on an external chip. See the “Hardware bridge” subchapter, below, for details.

usbp_perfmeas.h declares the performance measurement functions.

usbp_hwexport.h contains declarations of functions that will be visible at the application level. The hardware abstraction is visible, usually, only to the protocol part, but there are a few exceptions. Pipe activation and data transfer functions are visible to the application, to allow fast access to USB data as well as to decide when and whether a pipe needs reactivation. For the entire run of these functions, the microcontroller's USB interrupts are disabled. USB interrupt enable/disable functions are also available to the application, to allow it control over interrupt generation and servicing (if interrupts are configured as used, via compile switches).

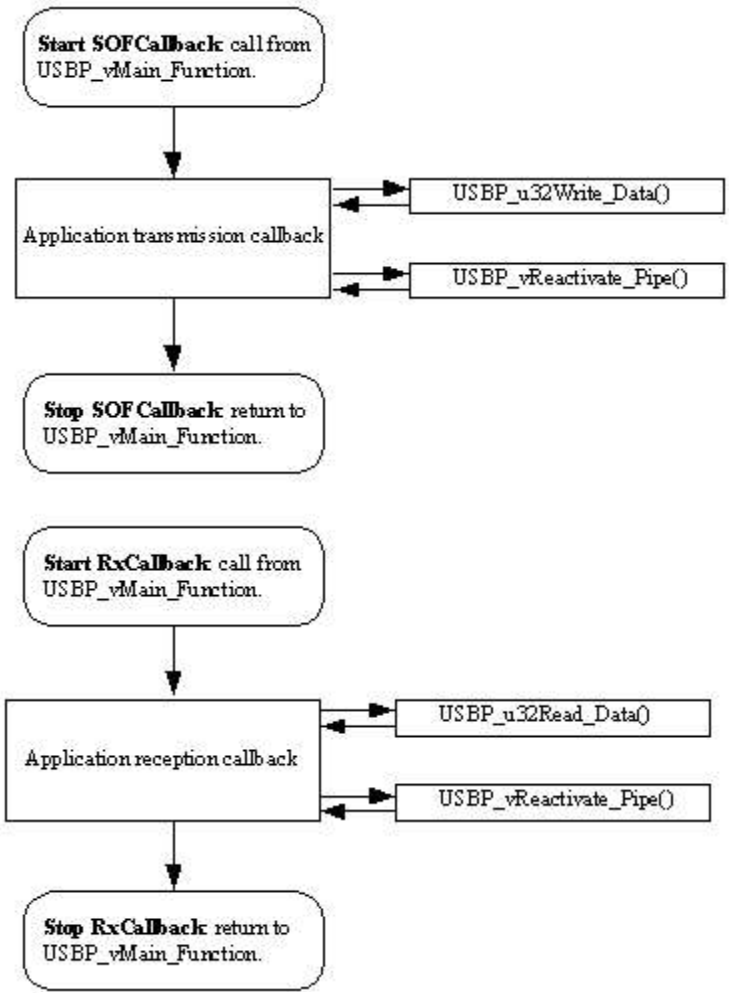

Fig 3.6 Accessing the USB hardware in callbacks.

A c file corresponds to every header, with the exception of usbp_hwexport.h. All declarations in this file are implemented in usbp_ucname_hwaccess.c.

The contents of any of these files may change during portation from one platform to another.

At its most simple, the hardware abstraction simply consists of sequences of accesses to special function registers that control the USB: defining where and what to write or read. Some other issues related to the hardware abstractions are presented in the following subsections.

Hardware bridge

Many microcontrollers have embedded USB hardware, and in this case it is usual to access the USB via special-function registers. However, some controllers do not have on-chip USB and must rely on an external USB transceiver chip.

NOTE: USB-to-UART and USB-to-LPT transceivers are NOT what is referred to in this document as “USB transceivers”. USB-to-UART or -LPT chips act as an already configured device (they pretend to be a modem or printer respectively), whereas a true “USB transceiver” contains no device configuration data and relies on its commanding microcontroller to provide the device configuration.

A true USB-transceiver is connected to the microcontroller typically via an SPI bus. This bus however is often shared with other resources like DACs or SPI Flash/EEPROM, and control over it may need an OS driver to ensure no conflicts appear. The interface between such a driver and the USB device firmware package can be implemented in the usb_hwbridge.c file and declared in the usb_hwbridge.h file.

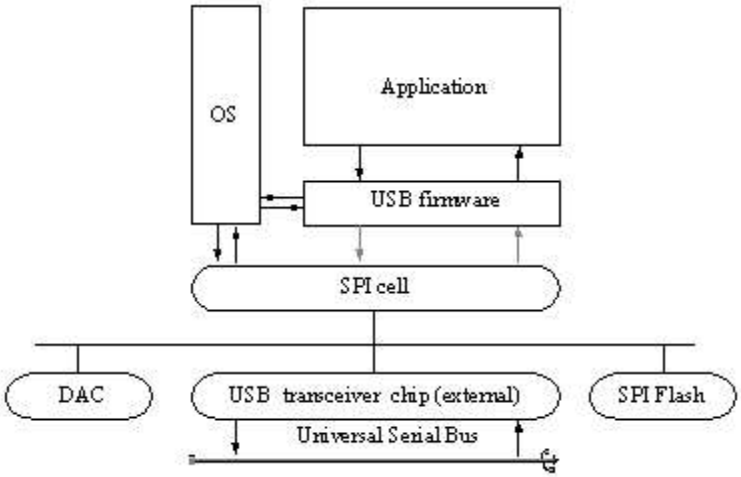

Fig 3.7 External transceiver chip; firmware uses OS calls to access the USB.

A special requirement placed on the bridge interface is that its operations be synchronous. That is: when a “read chip buffer”operation returns, the data is available; when a write operation finishes, the data has arrived at the USB transceiver chip. Operations that the USB transceiver chip performs (like sending data to the Host) can be asynchronous.

The bridge interface is only needed if the system uses a USB transceiver chip, or if for some other reason access to the USB hardware must be performed with the mediation of the OS. Otherwise, its files can be left empty.

Performance measurements

Useful in the development phase, performance measurement enables the measurement of the time spent by the USB_vMain_Function in total, as well as the sum of the time spent in communication and SOF event callbacks.

Performance measurement functions must access some time measurement resource on the controller, in which case they provide results directly. These results are visible at the application level, and can be reported by USB transfer. This means that the performance value received now was valid for the previous run of the USBP_vMain_Function). This is only an issue on systems using polling, where the previous run was likely callback-free and therefore not illustrative of a useful load. In order to prevent loss of callback time measurements, callback performance monitoring should be enabled and results only kept when callbacks are also activated.

Fig 3.8 Performance measurement with internal timers

Another strategy is to allow the performance measurement functions to control one or several IO pins, and then these pins can be monitored by an oscilloscope to ascertain performance. This has the advantage of being able to show at once, with no USB communication, the time spent in various parts of the program. It may also provide very accurate, sub-microsecond resolution, measurements. The disadvantage is the need for an oscilloscope.

Whatever strategy is used, the semantics of the measurement is as follows: a measurement is started when USBP_vMain_Function starts, and it will be finished when USBP_vMain_Function ends. Separately, measurements for time spent in callbacks (SOF and communication) is added by measurements started and ended around each callback. The actual time for USBP_vMain_Function is therefore the total minus the callback time.

The performance measurement functions are only specified as safe to be called the way they are employed by the USB package. They are not specified to be provided for the application for general purpose performance measurement, and are not visible at application level. Measurements should be reported in microseconds or smaller units of time, if meaningful compared to the measurement resolution.

Hardware portation

The detailed portation procedure is given in the “USB Hardware Portation Guide”. In brief, it is as follows:

- decide on whether the system needs a bridge interface or not;

implement it if it does.

- check consistency of type definitions from usbp_types.h with their names.
- at the very least, port a core set of functions to the new hardware (initialisation, status query, reset handling, pipe enabling/disabling, data access, send empty, send stall, send data package) as well as all hardware definitions like register addresses and bit masks that those functions need. This enables a minimal system to be built and tested; any not-yet implemented functionality should be disabled by compile switches, or empty functions should be supplied instead.
- (once a running system is obtained with the minimal function set) port the remaining hardware handling functions (interrupts, SOF tracking etc); optionally, port performance measurement. Any other useful register definitions should be added.

3.1.4 Configuration

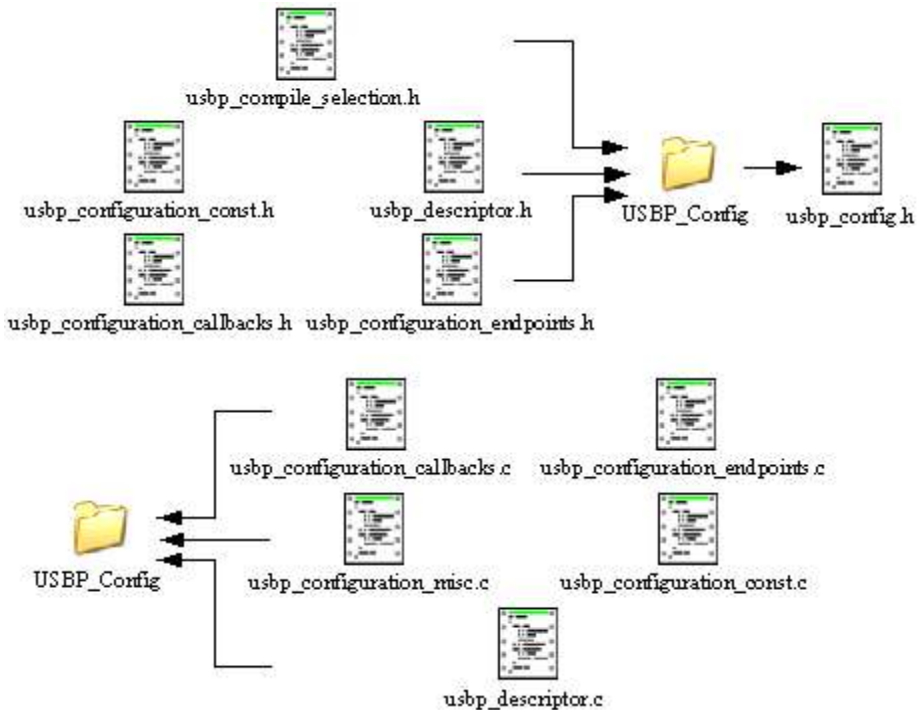

Fig 3.9 File structure detail: configuration

usbp_config.h gathers all the configuration headers in one file. It is usbp_config.h that is included by the rest of USBP.

usbp_compile_selection.h contains a set of compile switches that influence what code gets compiled or leaved out: multiple interface device, interrupt handling, SOF tracking, which events and requests trigger a notification to the application, which requests can be vetoed, whether performance measurement is used and whether callbacks are measured separately, whether to perform index checking when accessing

pipes etc.

usbp_configuration_callbacks.h declares all the used callbacks to the application: communication event callbacks and, if selected via compile switches, bus event and request notifications.

usbp_configuration_endpoints.h declares values for configuring the communication pipes: how many they are, which directions they have, maximum packet size, address of pipe buffer etc.

usbp_descriptor.h declares the descriptor variables: device, configuration, string, and any other used descriptor for example HID report.

usbp_configuration_const.h contains declarations that are to be kept unchanged from project to project.

With the exception of usbp_compile_switches.h, all configuration headers have a corresponding c-file with implementations of the declared variables and functions. usbp_configuration_misc.c contains variables for initial interrupt settings and remote wakeup enabling.

Callback writing

USBP has two types of callbacks: event callbacks and request notifications. Event callbacks are used for bus and communication events and are functions without parameters or return values. They simply inform the application that a particular event occurred.

Request notifications pass several parameters to the application, enabling it to identify the request. The application can then return a value that, if zero, signals to USBP that the application wishes to veto this request. The veto will take effect if USBP is configured to allow it.

Event callbacks can be used to read data from the host (reception callbacks), prepare new data to send to the host (transmission or SOF callbacks), synchronize the application to the USB (SOF callback) etc.

Request callbacks can be used to, for example, (re)initialize the data to be sent to the host after a SET INTERFACE request, or inform the application that it must change the format of data it sends and expects etc.

During any of these callbacks, the USB interrupt must be kept disabled.

Using the SOF and Tx callbacks for data transfers

Because the SOF will precede any data transfer in a frame, the appearance of a SOF on the bus is a good time to prepare outgoing data (from device to Host). The old joystick firmware used this strategy[3.2], and it is particularly useful for devices that use any transfer type except Bulk or Control, connected to a full speed bus.

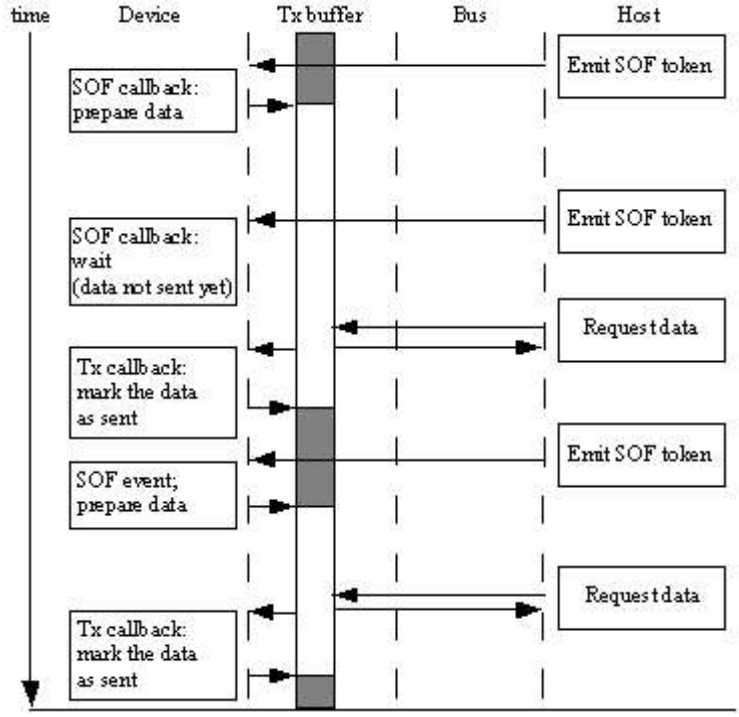

Fig 3.10 Using the SOF callback to prepare outgoing data

The diagram above illustrates a SOF-based data preparation mechanism. Care must be taken to the fact that the Host may not request data in a frame, for various reasons (system busy being a very common one). Therefore, new outgoing data must only be prepared if the old outgoing data was sent. It comes to the Tx callback to notify the application that, on the next SOF, it should prepare new data. It is possible, when using this mechanism, to prepare and send data in the same frame.

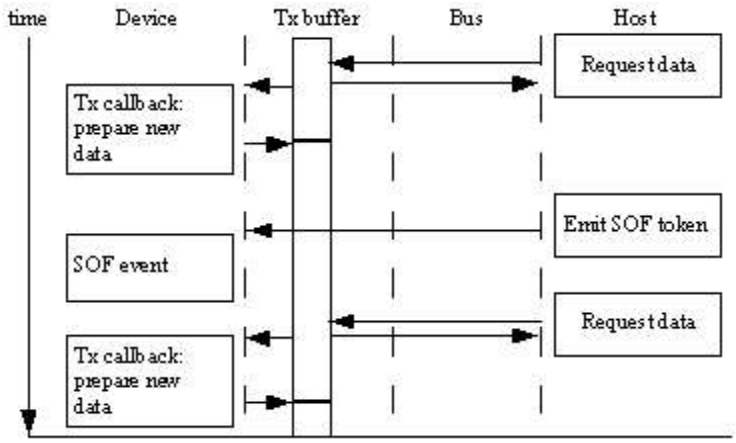

Fig 3.11 Using the Tx callback to prepare outgoing data

The diagram above illustrates an alternative: preparing the outgoing data on a Tx callback. This is also workable. However, on full speed buses it has the drawback of inserting an extra delay in the transmission, as the outgoing data prepared in one frame must wait at least for the next frame before being sent. Note that the software running on the device can ignore the SOF, and this strategy will still work. It can function also on high-speed buses with endpoints sending data several times per frame.

Bulk and Control transfers may happen at the end of a frame, and these transfers do not impose any timing on the delivery of data anyway, so which of the two strategies is used is not an issue with these transfers.

On high-speed buses, if an endpoint must send data more than once in a single frame, then the SOF-based strategy is not workable on its own. A mix of the two strategies would probably be optimal.

System configuration and integration

Configuring USBP for a given project consists of writing the appropriate values in each of the configuration headers and c-files. The detailed step-by-step procedure is given in the "USBP Configuration and integration" document, but in brief it is as follows:

- decide which compile switches to enable for a project. Of particular importance: shall interrupts be used, or multiple interfaces?
- declare (and at application level define) all callbacks that USBP will use. Communication events must have callbacks on each pipe. Callbacks on bus events and requests are optional.
- define pipe settings for every interface that the device provides. Usually only one interface is provided, but this need not be the case.

- write the descriptors (device, configuration, string etc). Some constants to assist in this process are provided, but this step requires some familiarity with USB descriptors. See chapter "9.5 Descriptors" of the USB Specification, revision 2.0, for more information.
- define interrupt settings.
- if necessary, write class code for the project.
- include usbp_types.h and usbp_driver_main.h function and find insertion points for USBP_vInit and USBP_vMain_Function. Assuming the hardware abstraction for the platform exists, USBP is ready.

3.1.5 Classes

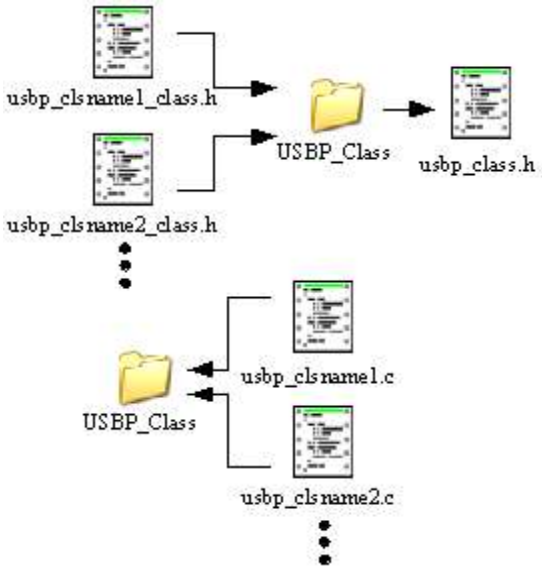

Fig 3.12 File structure detail: class handling

usbp_class.h includes all class headers. Other parts of USBP include this header instead of each class header. this allows for several classes to be defined, as well as for the class files to change names.

Each class header should have a corresponding c-file. While several classes may coexist, it is important that one of the class files provide three functions visible in the rest of USBP:

- USBP_vClass_Init(): initializes the classes, and starts one of them as the initial class. Called by USBP_vInit.
- USBP_vClass_Handle_Request(): is called by USBP_vMain_Function when the USB host issues class requests.
- USBP_vClass_Switch(): is called by USBP_vMain_Function when

a SET INTERFACE request occurs, and allows a previous active class to be replaced by the class of the new interface.

Typically, a device will only have one device class, or one interface class. The class-specific code may change according to project. Classes may need to define and use their own application callbacks, configuration variables and compile switches. These are not included in the USBP Configuration part, but in the classes themselves.

Vendor-class devices can function without any class code (except the three functions mentioned above, that are used by USBP in other parts; these functions can be left empty in this case). Currently, only the HID class is (partially but functionally) implemented.

4 The netX-based ROKVISS joystick

ROKVISS is an experiment run by DLR to test the feasibility of telepresence solutions for space applications. The experiment consists of "a two joint manipulator [that] can be operated from the ground by a direct radio link" [4.1]. The human operator controls the manipulator through DLR's High Fidelity Force Feedback Joystick [4.2], referred hereafter in this document as the ROKVISS Joystick.

4.1 The netX chip

Produced by Hilscher, the netX chips are a family of high-end controllers with support for a multitude of network and communication protocols. At the time of this diploma work they are relatively new, but promise to be of great use in a large range of automation and robotics applications. Because of the high integration of the communication features, the ROKVISS joystick was updated with a new, netX-based control electronics, as part of a previous diploma work (Matthias Faehse, "Entwicklung einer Steuerelektronik fur den DLR kraftreflektierenden Joystick", February 28, 2007).

4.1.1 Features overview

A summary of the features of netX is given below[4.3]:

- ARM core, running at 200MHz.
- 2/10/100 Mbit/s Ethernet; support for Real-Time Ethernet protocols: EtherCAT, Ethernet/IP, Powerlink, PROFINET, SERCOS-III.
- Fieldbus controller, supports PROFIBUS, CAN, Interbus.
- full-speed USB interface, can act as host or device.
- SPI, two quadrature encoders, two AD converter channels on 10 bits with 4 channels each, three-phase PWM
- JTAG interface
- IEEE 1588 compliant system time measuring

4.1.2 EtherCAT

EtherCAT is an Ethernet-based communication protocol that allows good bus utilization. It uses a Master/Slave principle, where the Master can be any device capable of Ethernet communication. It sends an Ethernet frame to the first Slave in an EtherCAT segment, which will react on it, extract and/or insert data, then send it to the next Slave and so on until the last Slave processes the frame, and it is sent back, via the first Slave, to the Master[4.4]. The Master can be a PC with an Ethernet card; the Slaves are controllers with special EtherCAT hardware.

There is an EtherCAT stack provided for netX, and it remains as a future development to use this stack in a real project, like the ROKVISS joystick. On the Master-side, several pieces of software tools can be used to sustain EtherCAT communication, for example the TwinCAT suite from Beckhoff Automation GmbH.

4.2 The control board

Developped by Matthias Faehse during the course of his diploma work [4.5], the control electronics is built around the netX chip. It has a USB and two Ethernet connections; additionally, debug via JTAG is provided.

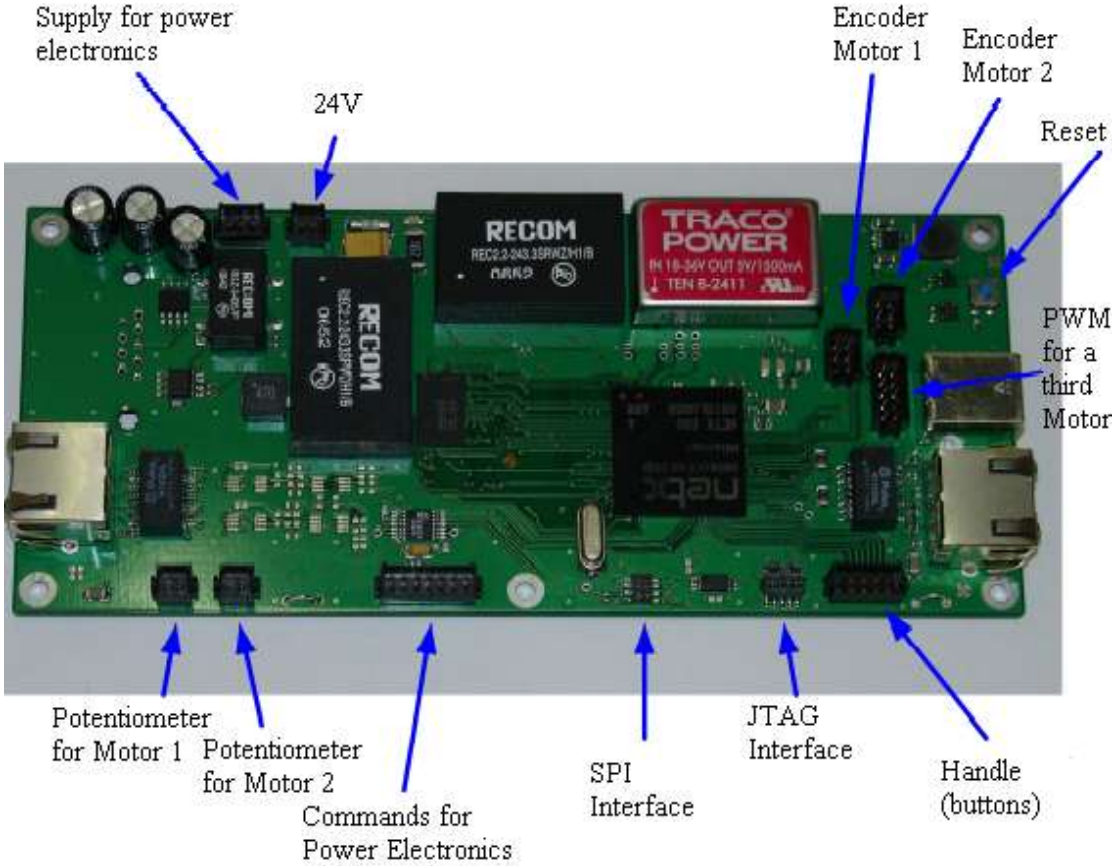

Fig 4.1 The control electronics; original from [4.5], labels translated

The control electronics board can read the status of several buttons, connected to digital inputs of the netX. It can measure several voltages with the aid of netX's ADCs. PWM control is left to a different board. The PWM value is commanded through a voltage output by a DAC, connected to the netX via SPI. Several digital outputs are also used to command motor directions. Position is measured through quadrature encoders. Two motors can be controlled and have their position monitored, and PWM signals for a third motor are generated.

4.3 The joystick software

4.3.1 Functionality

During the course of this diploma work, the software on the old joystick was ported to the new control electronics. The joystick software is capable to measure the position of the joystick, send this measurement back via USB, and relay torque commands to the motors. In addition to the old functionality, round-trip time measurement and performance measurement for the USB firmware is provided.

To fulfill its functions, the joystick software needs to work with several hardware modules inside the netX: GPIO, ADC, quadrature encoder interface, SPI, USB and system timer. The first three of these have been developed during Matthias Faehse's thesis, the others have been developped for this diploma work and will subsequently be presented.

The SPI “drivelet”

Because of the large code overhead, the SPI driver that comes with the rcX operating system was avoided, and instead a much more limited (but for the purpose of this application, sufficient) driver was made.

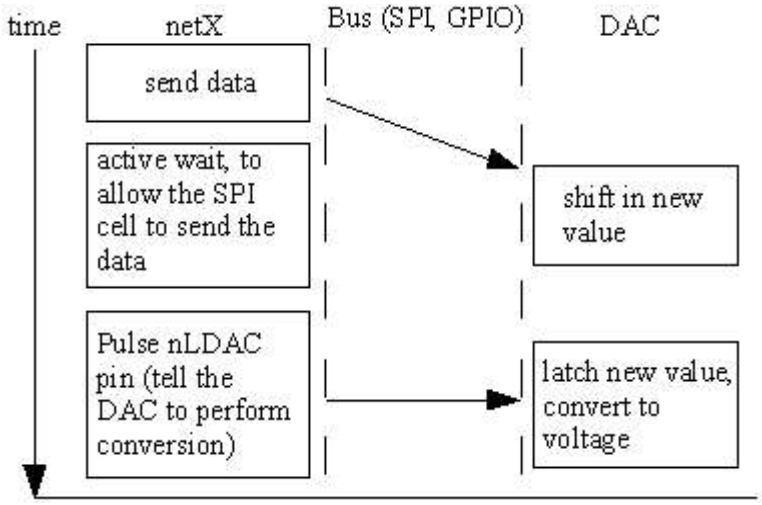

Fig 4.2 Commanding a voltage output via the DAC

The “drivelet” will only send data on SPI, since the DAC only needs to receive data from the netX, it does not need to send data back. No interrupts are used, active delays (about 25 microseconds) ensure that enough time is given for the SPI transmission. A general purpose I/O pin from the netX, connected to the DAC, is briefly set to 0 to signal the DAC that a new conversion must be made.

An issue with the netX SPI cell is that data must be byte-swapped before sending on the SPI. A sample macro for byte swapping is given below:

```
#define ByteSwap(intVal) ( ((intVal&0xFF)<<8) | ((intVal>>8)&0xFF) )
```

netX has said that this feature of the SPI will NOT be changed, because several applications have already been built that use byte-swapping.

The system timer

netX provides a counter that is incremented each 10 nanoseconds. It allows measuring the system time with this resolution; with the "seconds" and "nanoseconds" registers giving the full system time[4.6].

Getting the counter value is straightfoward, the registers simply need to be read: the "seconds" register must be read first. Since the counter runs authomatically, no initialisation or reactivation is necessary.

The system timer was used both for round-trip time and performance measuring. The time elapsed between two reads of the system timer can be calculated by simply substracting the values obtained at the two reads. Note that the "nanoseconds" register counts to a billion and then resets to 0.

The USB firmware: USBP

The netX joystick is the first project where USBP was deployed. While the general characteristics of USBP were discussed in their own sections, this chapter will overview the specific hardware platform and configuration aspects.

netX has an internal "On-the-Go" USB controller, which means that it can act as either a Host or a device. For the joystick software, only device functionality was needed. The USB controller being internal, no special protocol is needed to access it. Therefore, the "hardware bridge" part of USBP is left empty.

Full-speed USB operation is guaranteed, and up to 8 pipes can be configured. Accessing the pipe configuration bits requires first setting a pipe index value in a "pipe selection" register; it is not the case that each pipe has separate registers. Pipes do however have different data buffers, which are visible as RAM memory areas.

The hardware abstraction level of USBP for netX was developped using the "netX Program Reference Guide"[4.7], and in general the USB controller performs as

expected. The only problem is that if a USB connection between netX and the PC exists when the netX is reset, then after the reset completes netX will not re-establish the connection. This is a hardware issue[4.8], and the work-around is to break and replug netX after a reset. Another solution is to connect the USB data lines to pull-ups controlled by netX's GPIOs [4.8], but this option is not available on the joystick's control electronics.

The configuration part of USBP was tailored for the application: two communication (for transmission and reception respectvely), SOF and SET CONFIGURATION callbacks were defined. Endpoint settings were selected: pipe directions, packet sizes, transfer types. Descriptors were written to declare the device as an HID, for connection to a Windows system. The interface class in the descriptors was later changed to Vendor for connections to Linux/QNX.

The joystick software

A logical diagram of the joystick software is given below.

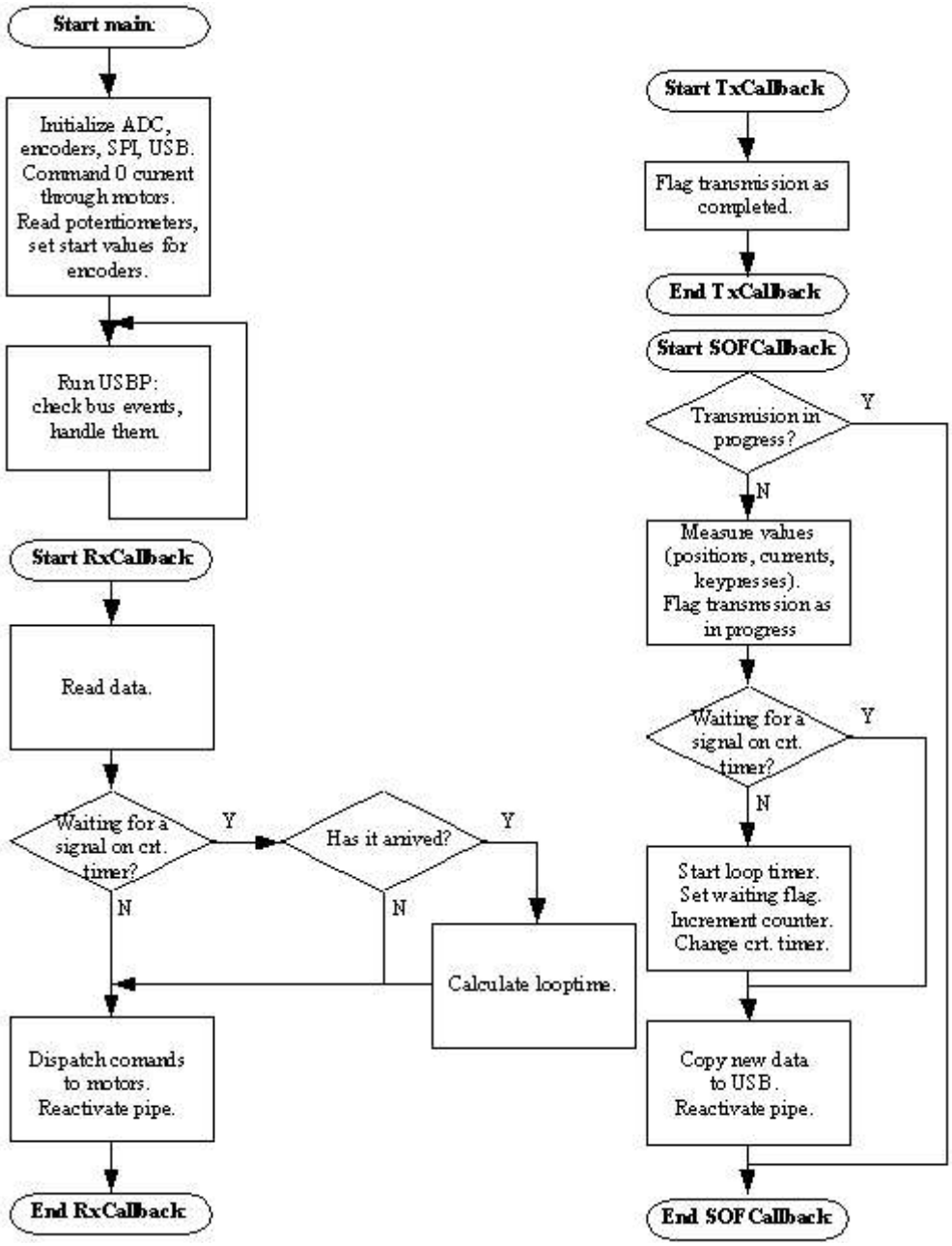

Fig 4.3 Logic diagram of the netX application

The main function simply initializes the various parts (USB, encoders etc), and then enters an endless loop where the USB is polled. If a SOF event occurred, and the previous transmission was completed, a new packet of data is prepared. A transmission even triggers the Tx callback, which is used to flag transmissions as completed. A reception event triggers the RxCallback function, which will read and then dispatch to the motors the commands sent by the PC. Loop timing code is also present in these two callbacks.

The transfer packets have the following structures:

Member name	*Usage*
u16CtrlState	signal for loop timer
i16MeasPosX	X-axis position (encoder raw value)
i16MeasPosY	Y-axis position (encoder raw value)
i16FMeasX	X-axis motor current (ADC raw value)
i16FMeasY	Y-axis motor current (ADC raw value)
u16TastStat	button statuses (0 – pressed)
u16LoopTime	loop time (microseconds)
u16MainTime	total time spent in USBP_vMain_Function (microseconds)
u16CallbackTime	total time spent in communication callbacks (microseconds)

Tbl 4.1 Transmission packet: typedef struct typMeasVal

Member name	*Usage*
u16CtrlState	signal for loop timer
i16SollPosX	X-axis desired position (not yet used)
i16SollPosY	Y-axis desired position (not yet used)
i16FSollX	X-axis commanded force (milliN)
i16FSollY	Y-axis commanded force (milliN)

Tbl 4.2 Reception packet: typedef struct typCmdVal

ADC “raw values” are the exact values read from the signal converter. These values can be transformed into more convenient units of measurement, like radians or degrees, on the PC side.

The motor commands will be integers in the -10000 ... +10000 range, where each increment represents 1 mN, and give the force expected to be exerted at the point immediately below the button-area of the handle.

The joystck was calibrated by measuring the forces output at that point on the joystick when the DAC was issued one from a set of commands. Each axis had 14 points measured for the positive and 14 points for the negative direction. Based on these points, a look-up table with interpolation allows the joystick to convert mN into values to send to the DAC.

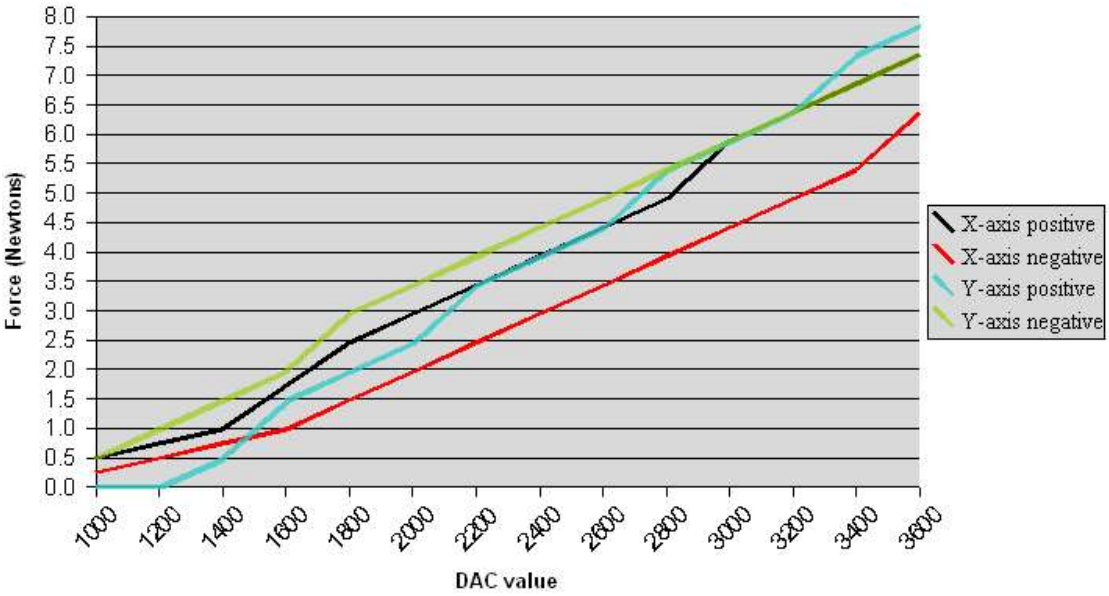

Fig 4.4 Motor calibration measurements

A special note: the next command above 3600 issued to the DAC resulted in the motor sinking a large amount of current, but not outputting much torque. Therefore, DAC commands are limited to 3600, and the maximum forces that can be commanded are about circa 7N.

4.4 Performance of firmware

Using a map file generated from building the joystick software with an evaluation Hitop toolchain, and summing all bytes needed by USBP files, the following results are obtained for USBP code and data sizes:

Files\Memory sections	*text*	*rodata*	*data*	*bss*	*common*
usbp_driver_main.c	2716	428	0	0	96
usbp_netx_hwaccess.c	0	0	2	0	12
usbp_netx_perfmeas.c	340	0	4	1	24
usbp_descriptor.c	0	286	0	0	0
usbp_config_callbacks.c	4	24	0	0	0
usbp_config_endpoints.c	0	28	0	0	0
usbp_config_misc	0	0	1	0	0
usbp_netx_register_defs.c	2740	108	0	0	0
usbp_config_const.c	40	0	0	0	4
usbp_hid_class.c	172	0	0	0	2
Total	6012	874	7	1	138

Tbl 4.3 Amount used from memory sections (bytes)

Performance measurements show that USBP_vMain_Function requires 6 microseconds to complete (this is the difference between total and callback time).

There are wait-states when accessing the USB hardware. For example, the USBP_vGrab_Status_Bits function, consisting merely of two reads and two writes to USB hardware registers, uses 2.5 microseconds. The value was obtained by using performance measurement routines placed around this function. It follows that minimizing the number of accesses to USB registers is desirable.

4.5 The joystick demo application

The previous version of the ROKVISS joystick had a demonstrational application running on LINUX, to show the joystick's force feedback capabilities. The same application was reused in this diploma work for the netX-based joystick, with only three modifications: the communication strategy (discussed in chapter “5.2 Linux - libusb”), integrating loop time measurments in the application, and the addition of keyboard input.

For the new loop time functionality, the application will read the signal value (u16CtrlState) and send it back unchanged to the netX. It will also read and then calculate the average of round-trip time values sent by the netX.

Keyboard input was added because at the time the joystick was not equipped with buttons. Now, either pressing a key or a joystick button will perform various

functions, like changing the current scenario, or changing the value of one of the scenario's parameters.

Key	***Usage***
Y y	previous scenario
C c	next scenario
W w	previous scenario parameter
S s	next scenario parameter
D d	increment current scenario parameter
A a	decrement current scenario parameter
L l	set down-left corner position
P p	set up-right corner position
Q q	quit program

Tbl 4.4 Key input to the joystick demo application

The scenarios consist of the following:

- "gummiband": the joystick moves an elastic band across the screen. The band can catch a ball, lift and throw it, and the elastic force appearing in the band is output on the joystick. Contact between the band and the yellow box at the top left is also feelable in the joystick.
- "sphere": the joystick moves a yellow ball on the screen. At the centre is a large and fixed cyan ball. Contact between the two balls can be felt in the joystick.
- "billiard": the joystick moves a magenta ball on the screen. Impacts with the other balls or with the edges of the screen can be felt in the joystick.

All collisions in the program are modelled as elastic, with some viscous damping. The diameter of the balls in scenarios "gummiband" and "billiard" are proportional to their initial masses. Some masses can be changed during a scenario's run.

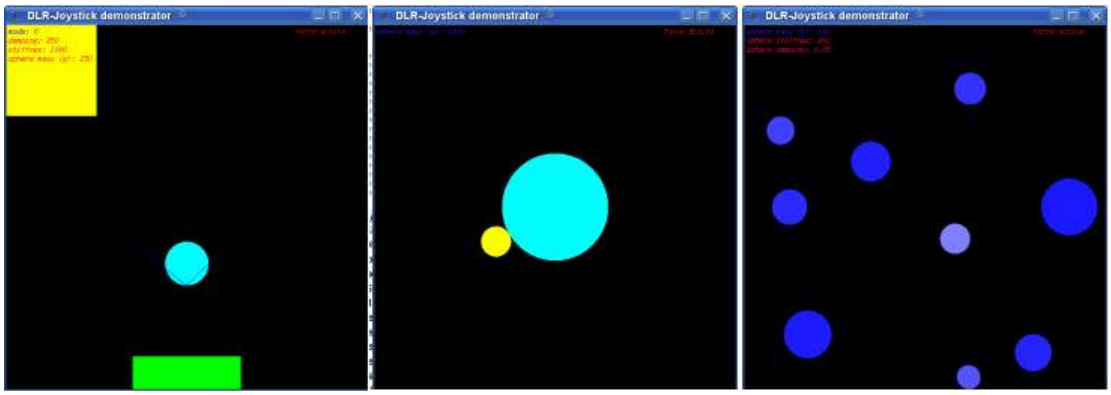

Fig 4.6 Joystick demo application: scenarios

5 The USB device drivers

On the host side of the USB connection, there must exist functional parts that can handle a USB device- typically, a software driver. It is the role of this driver to provide an interface for an application program running on the host to the USB host controller driver, and allow the application to detect that a certain device has been connected, work with that device, detect when it is disconnected. Most often, while a USB device driver is not part of an operating system kernel, it nonetheless is made to operate in kernel-space and is a kind of OS extension.

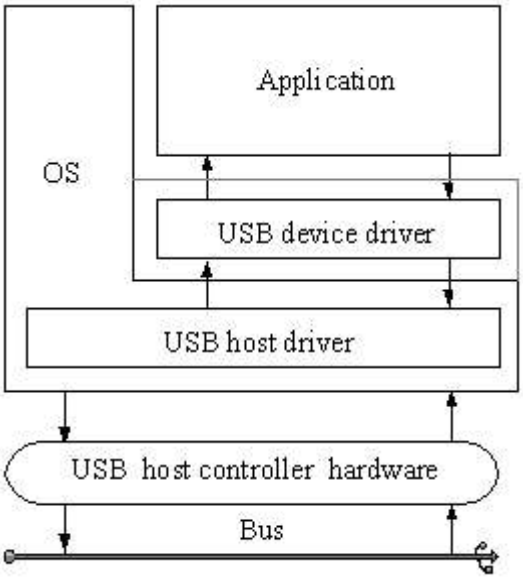

Fig 5.1 Software hierarchy on the Host

In this chapter I will present the “drivers” made and/or used during this diploma work on various operating systems: Windows XP, Linux, QNX. The title “driver” is a bit of a misnomer because most of these software pieces are not the typical kernel-space driver, but they nonetheless provide the same service to the application.

5.1 USB communication overview

The next diagram illustrates a typical information and request flow on the USB bus. Note that it is always the Host that initiates data transfers. The SOF token will be generated periodically by the Host hardware.

In this example, the Host has nothing prepared for the first frame. The device driver on the Host allocates two transfers (one transmission, one reception) which will take effect on the next frame. On the device side, the device prepares data for transmission when the SOF event is detected by the hardware.

On the second SOF, the device will not prepare new data, because the old packet was not sent yet. However, during this second frame the Host will request data from the

device, which will trigger on the device side a transmission event. Data will be sent from the device's USB hardware buffer, and the device software will be notified that a transmission took place. The Host will also send data to the Host (triggering a reception event on the device side). As the read and write operations complete, the USB driver on the Host notifies upper layers (the device driver software on the Host), and transfers for the next frame are allocated.

On the third SOF, the device prepares new data because a transmission happened in the previous frame. The rest of the frame goes on just as the second one described in the paragraph above. Ideally, all subsequent frames will be identical to the third frame: the device will prepare data on the SOF (because the previous frame contained a transmission), the Host will request the prepared data and send some data back. Therefore, the shortest time needed to complete a communication request, whether read or write, is one frame (one millisecond), which is the theoretical best sample and round-trip time.

One further issue to consider is that notifications on the Host side are not necessarily immediate, because they pass through many layers of software. This can add delays in communication. (On the device however, notifications are usually almost immediate,: taking as long as a call to an interrupt or a function on simple embedded systems).

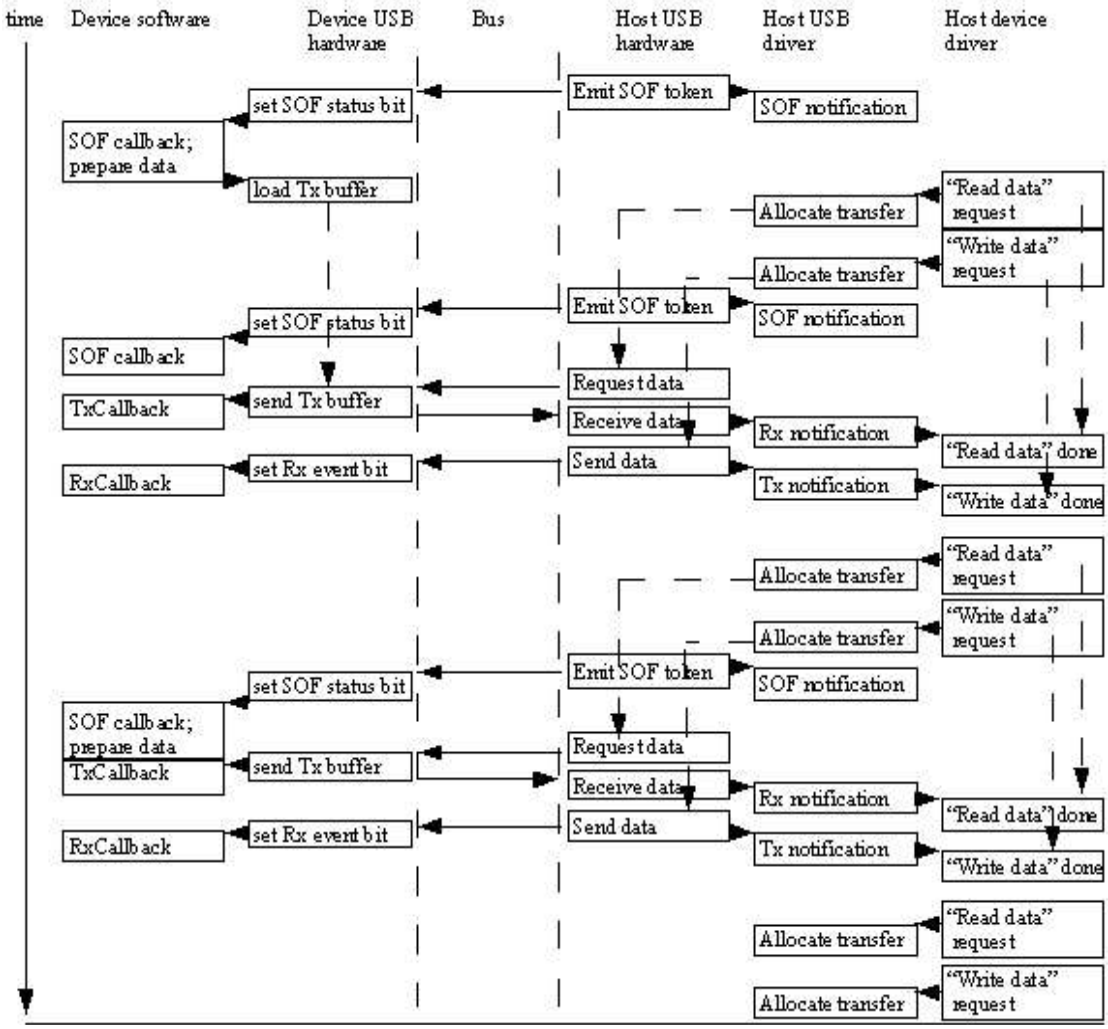

Fig 5.2 USB communication flow

5.2 Windows XP – the HID driver

For Windows the existent HID (Human Interface Device) driver was chosen to connect to a USB device, because this solution promised several key advantages. First, no new driver had to be written for the device. Second, the HID class is not only a standard class, it is supported already by many operating systems: Linux and QNX each have an HID driver. Third, the HID driver for Windows is quite well documented and using it is fairly easy. Fourth, despite some limitations, the HID class is theoretically sufficient for implementing simple process control applications. The relative simplicty of using HID as a class for USB makes it recommended for developing various USB peripherals for Windows XP [5.1], not all of them necessarily related to user interfacing. For example, voltmeters can be configured as HIDs [5.2].

Unfortunately, the documentation for the Linux or QNX HID drivers is incomplete or hard to find, and it seems that the Windows HID driver is much better performing

than its equivalent on Linux. Therefore, what promised to be a portable solution remained limited to the Windows family.

5.2.1 HID class: overview

Human Interface Devices are specified by the USB standard as either low- or full-speed devices that must have a control endpoint and an IN interrupt endpoint, that is they send data to the host. Typical examples are mice and keyboards. An HID may also have an OUT interrupt endpoint, which allows it to receive commands from the host[5.3]. For example, the LEDs on keyboards may be turned off or on by commands from USB.

Force feedback input devices can be covered by the HID, or by a special extension of the HID class, the Physical Interface Device class (PID). All PIDs are HIDs, and are controlled by the same driver[5.4].

Because of their limitation to full-speed and one pair of interrupt endpoints, HIDs can send at most 64 bytes each millisecond. They can receive the same amount of data from the Host at the same speed.

To be recognized as an HID, a device must provide an HID class descriptor and an HID report descriptor. The HID class descriptor gives the version of the class specification, the type and number of HID descriptors to follow- usually, one report- and the length of those descriptors. The HID report descriptor gives the structure in of the data packets: how many records, how long, what function they serve, for example commanding the Caps Lock LED or giving information on X-axis displacement.

Of these, the HID report descriptor is problematic, because it must be written consistently. For example, the number of declared in-going or out-going bits must match the number of bits allocated to declared uses. See chapter “6.2.2 Report descriptors” of the Device Class Definition for HID document for more information. A general recommendation is to start with a working setup and change it in small steps, testing each one of them.

The HID driver gives access to the raw data of a report. Therefore, the application may choose to ignore the uses allocated to the bits by the descriptor. However, the driver will reject reports if they are not consistent in byte length with the byte length calculated from the sum of all records declared by the descriptor, or the descriptor itself is not correctly written. In this case, the application would read no data from the device, and attempts to write to it will fail.

One final issue is device caching: the driver will “remember” the devices it saw connected by their description and serial number strings. If the report descriptor is modified, but none of these strings, then the driver will not request the report descriptor and not be aware of the changes. To avoid communication problems, every change in the report descriptor must be accompanied by a change in the string descriptors.

5.2.2 Using the HID driver: usbhidio

The source code for using the HID driver was taken from a free HID enumeration and testing utility, usbhidio_vc6, developped by Jan Axelson from www.lvr.com. It is meant to assist in developing and testing HID peripherals. With only a slight modification, it was used in this diploma work; annex "A5 Testing USB on Windows" describes how to build the source code. The Microsoft Windows Driver Development Kit (Windows DDK), free for non-commercial applications, is also necessary in order to build the code.

Accessing an HID is, from the point of view of the application, similar to accessing a file. Once the HID is found, a file can be opened to represent it. Reading and writing to this file exchanges data with the device. These operations can be overlapped, and therefore performed simultaneously (in the same USB frame).

One issue is to find the device. A loop enumerates all connected HID devices, looking for certain Vendor and Product Ids. If it finds them on a device, it opens file handles to that device. Once the handles are successfully opened, communication is possible.

The modification from usbhidio.c comes in the form of a file buffer flush performed through HidD_FlushQueue on the read file handle. With the original usbhidio code, there would be an increasing delay between the data the application sends to the device, and the data the device actually receives. The cause was likely the internal buffering inside the HID driver. Flushing the buffer assures that the delay between the moment that the device sends data, and the moment the application receives it, stays constant.

5.2.3 Running a USB communication in Matlab (Windows XP)

A Simulink model (joystick_test_WIN.mdl) was used to maintain a dialogue with a USB peripheral. A MEX function is used to interface Matlab to the HID driver. To assess performance, measurements of "round-trip time" are used.

Round-trip time is the time measured, on the USB device, between preparing a packet of data for the Host (usually, on a SOF event), and the moment when the response to that particular packet of data is received from the Host. Being much simpler and with fewer processes running, the device can measure time easily and accurately (up to microsecond resolution), with negligeable and/or constant delays between events on the USB and the reaction to them in software.

To identify data packets, the USB device sends a counter inside each of them; it also will start a timer upon doing so. The application on the Host will send the same value it receives for this counter back to the device. The USB device, on a reception event,

will- if the counter value received from the Host matches the expected one- stop the timer. The counter value is incremented when a new packet is prepared for sending (ideally, at each SOF event).

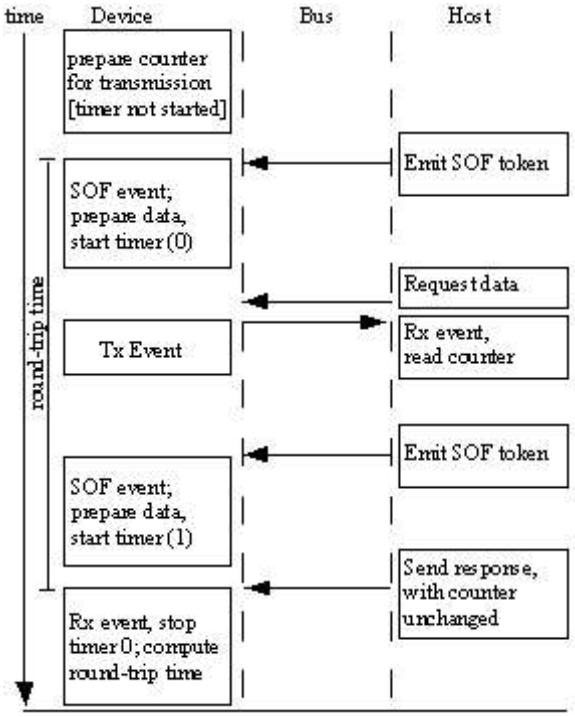

Fig 5.3 Measuring round-trip time

Two different timers (or pairs of measured time values, from the same timer) are used in alternation, because between the moment when a packet of data is sent and the moment when the answer to that packet comes back, another SOF takes place.

Data will be prepared on a SOF event only if the previous packet of outgoing data was sent. The counter is incremented only when a packet is prepared. A timer is only started if it was not already running. All these steps are necessary to ensure correct functionality even if the Host will not request data on each frame, and will delay response to packets more than the ideal situation presented in figure 5.2.

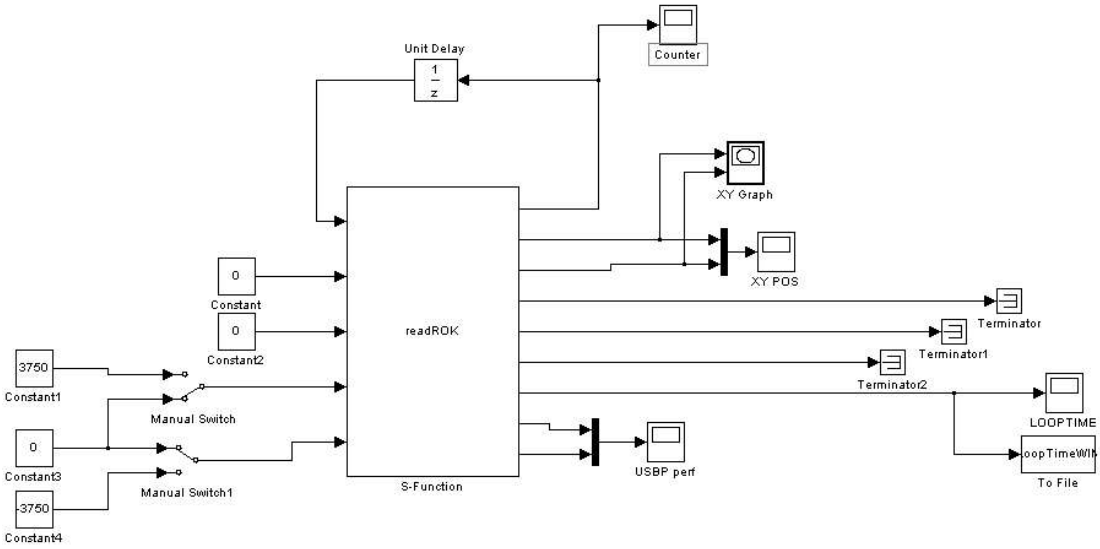

Fig 5.4 Simulink model: joystick_test

A snapshot of the simulink model is shown above. The XY Graph has been added to allow a graphic tracking of the joystick. Experiment has shown that NO XY plots should be used, if communication performance is desired. When the XY Graph is active, round-trip times show a tendency to increase, and start from double of what can be achieved without XY Graphs running.

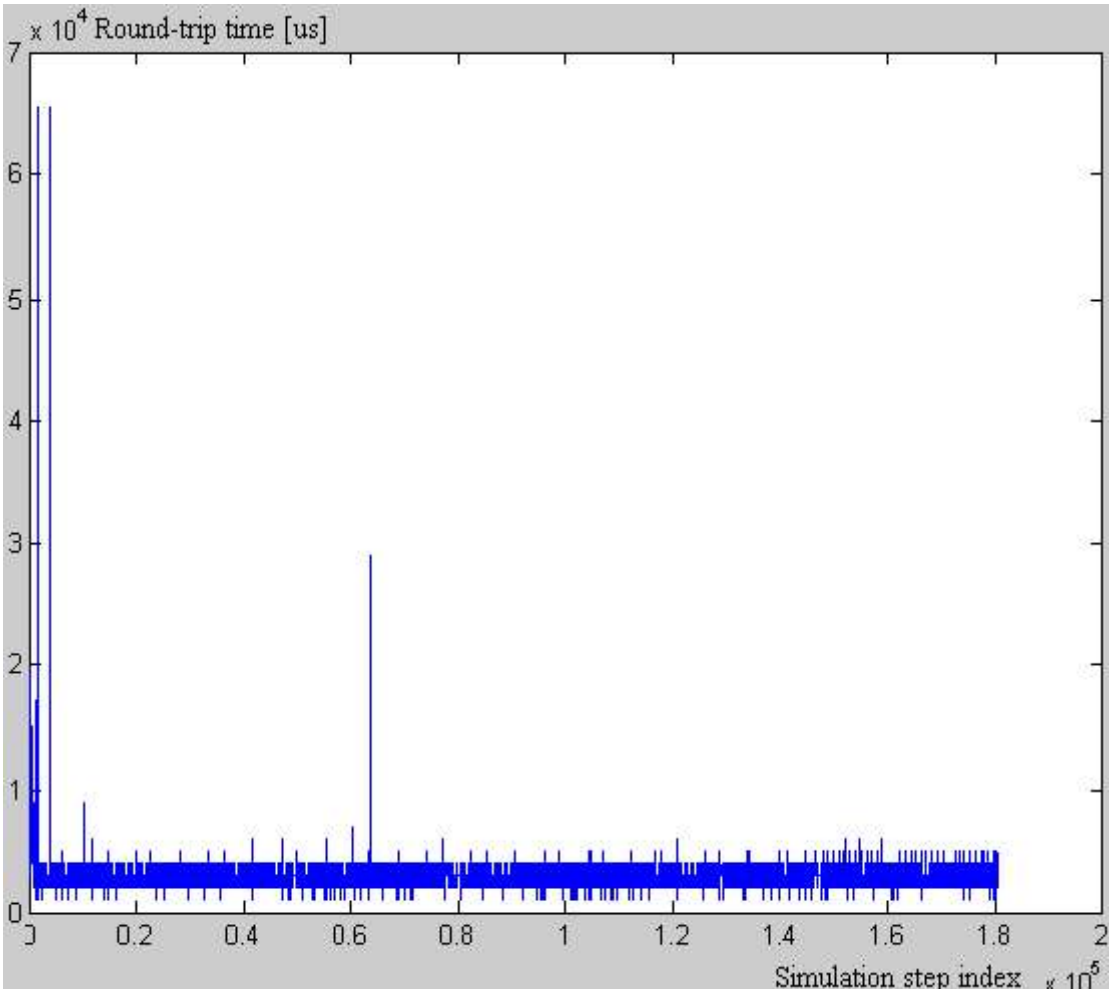

Fig 5.5 Round-trip time vs. simulation step plot: Matlab, normal mode, Windows XP

The figure above shows the loop time in microseconds with the model running in "normal mode" for six minutes. Running in "external mode" makes the model more resistant to events inside the Matlab window, but still not with real time performance. Round-trip time varies typically from 2 to 5 milliseconds, rarely getting as low as the theoretical best of 1 millisecond. A simulation step typically takes about 2 milliseconds of real time.

A few notes on this figure: the large spikes at the beginning are caused by the model updating an XY Graph, and are rare compared to smaller values. Closing the XY Graph, leaving only the looptime plot open, makes the large spikes disappear. After that the model exhibits usually from 2 to 5 millisecond round-trip time, but with very many spikes. The Matlab windows should be left alone: not resized nor moved, no changes to parameters.

5.2.4 Controlling position from MATLAB via USB

As a further test of the quality of USB communication in Windows, a small motor position control application was implemented. The netX microcontroller on the joystick will measure, via a quadrature encoder, the position of a motor, and relay

current commands to it. The program running on the netX for this application is very similar to the joystick program, except that in this case only one motor is controlled rather than two. It is described in annex “A3 Monitoring a motor with netX”

The control algorithm is carried out by Matlab, in a Simulink model described in annex “A5.2.2 Motor position control in Simulink”. The control algorithm communicates with the netX via USB.

The motor is a DC servo with a nominal supply voltage of 24V. Its electrical time constant[5.5] is 154.54 microseconds, and was deemed negligeable in this application. The mechanical time constant is 16 milliseconds. The moment of inertia, according to the datasheet, is 49g*cm^2. The motor's current-to-torque constant is 41mNm/A. The viscous friction was measured by putting a constant current through the motor and measuring the speed, and its value is 9.2mN*cm*s.

The quadrature encoder used to measure its position produces 4096 impulses per lap. netX contains a quadrature decoder hardware capable of counting 65536 impulses before rolling over. This range is extended via software to a 4-byte signed integer variable (32bits), to allow position to be tracked across more laps.

The current through the motor is controlled by a PWM with duty cycle set by a voltage output by a 12-bit DAC.

On the PC, the Simulink model will compare the motor position with a desired position (measured in radians), and calculate what current the motor should consume to reach that desired position. The control algorithm used is PI (proportional integrative). A repeated square wave is used as an input for the desired position, in order to assess response time and overshoot.

The motor will be modelled using a current-to-position transfer function [5.3]:

$$Hmot(s) = \frac{\Theta(s)}{I(s)} = \frac{1}{(J \cdot s^2 + d \cdot s)}$$

where J is the moment of inertia and d is the viscous friction (measured in A*s^2 and A*s respectively, by scaling J and d with the motor's torque-to-current constant). Electrical time constant is considered 0.

The motor PWM command circuit and encoder are modelled by constant-1 transfer functions. The raw value read from the encoder will be multiplied by a conversion constant to get the position in radians; the value sent to the command circuit is first multiplied by a conversion constant to change the amperes commanded by the control algorithm into a DAC command.

For a first step, the control algorithm is considered as time-continuous, and an added delay block gives a time-discrete character to the control (Ts is the round-trip time):

$$Hdel(s) = e^{-s \cdot Ts}$$

The control algorithm will be a position (angular) to current transfer function, and will have the expression:

$$Hpi(s) = kp_{pi} + \frac{ki_{pi}}{s}$$

The kp_pi and ki_pi parameters must be positive, and it is given that:

$$Ti = \frac{kp}{ki}$$

Because the control system will not provide a constant round-trip time, the phase margin is used to tune its parameters. This way it can gain some resistance to rare and relatively small increases in round-trip time. The larger the phase margin the more resistent, and also with less overshoot and fewer oscillations, the system.

The problem is to find the kp and ki, control parameters, given the motor parameters J and d, the control loop parameter (usual) round-trip time Ts, and some additional tuning related parameters like the desired phase margin.

The open-loop transfer functions obtained is simply the product of the control algorithm and the motor transfer functions:

$$Hol_{pi}(s) = \frac{kp_{pi} \cdot s + ki_{pi}}{J \cdot s^3 + d \cdot s^2} \cdot e^{-s \cdot Ts}$$

From the gain and phase expressions,

$$|Hol_{pi}(\omega)| = \sqrt{\frac{ki_{pi}^2 + kp_{pi}^2 \cdot \omega^2}{d^2 \cdot \omega^4 + J^2 \cdot \omega^6}}$$

$$Arg(Hol_{pi}(\omega)) = -\omega \cdot Ts + atan\left(\frac{kp_{pi} \cdot \omega}{ki_{pi}}\right) - (\pi + atan\left(\frac{J \cdot \omega}{d}\right))$$

and imposing conditions that gain be one at a frequency ω_g and phase margin be φ_{mg}, the subsequent expressions

$$(ki_{pi} = \frac{kp_{pi}}{Ti_{pi}}) \qquad kp_{pi} = \sqrt{\frac{d^2 \cdot \omega_g^4 + J^2 \cdot \omega_g^6}{\omega_g^2 + \frac{1}{Ti_{pi}^2}}}$$

$$\varphi = atan\left(\frac{kp_{pi}\cdot\omega_g}{ki_{pi}}\right) = \varphi_{mg} + \omega_g\cdot Ts + atan\left(\frac{J\cdot\omega_g}{d}\right) \quad (0 \leqslant \varphi < \frac{\pi}{2}) \quad Ti_{pi} = \frac{tg(\varphi)}{\omega_g}$$

permit calculating the control parameters for the PI algorithm. If φ does not respect the condition in paranthesis, then it is not possible to obtain the desired phase margin at the given frequency.

Bode plots of the resulting system are produced, in order to check that the imposed phase margin is indeed obtained, and that the gain margin is positive.

Finally, the control algorithm is discretized using a bilinear transform, to yield the following expression, which can be implemented in the Simulink model:

$$y[n] = x[n]\cdot\left(kp + ki_{pi}\cdot\frac{T_s}{2}\right) + x[n-1]\cdot\left(ki_{pi}\cdot\frac{T_s}{2} - kp\right) + y[n-1]$$

where y[n] is the command (current) at step n, and x[n] is the difference between prescribed and actual position at step n.

The figure below shows the motor control behaviour with a square wave used as the prescribed postion value. The open-loop system was tuned to have a 60^o phase margin, and gain one at 7rad/s.

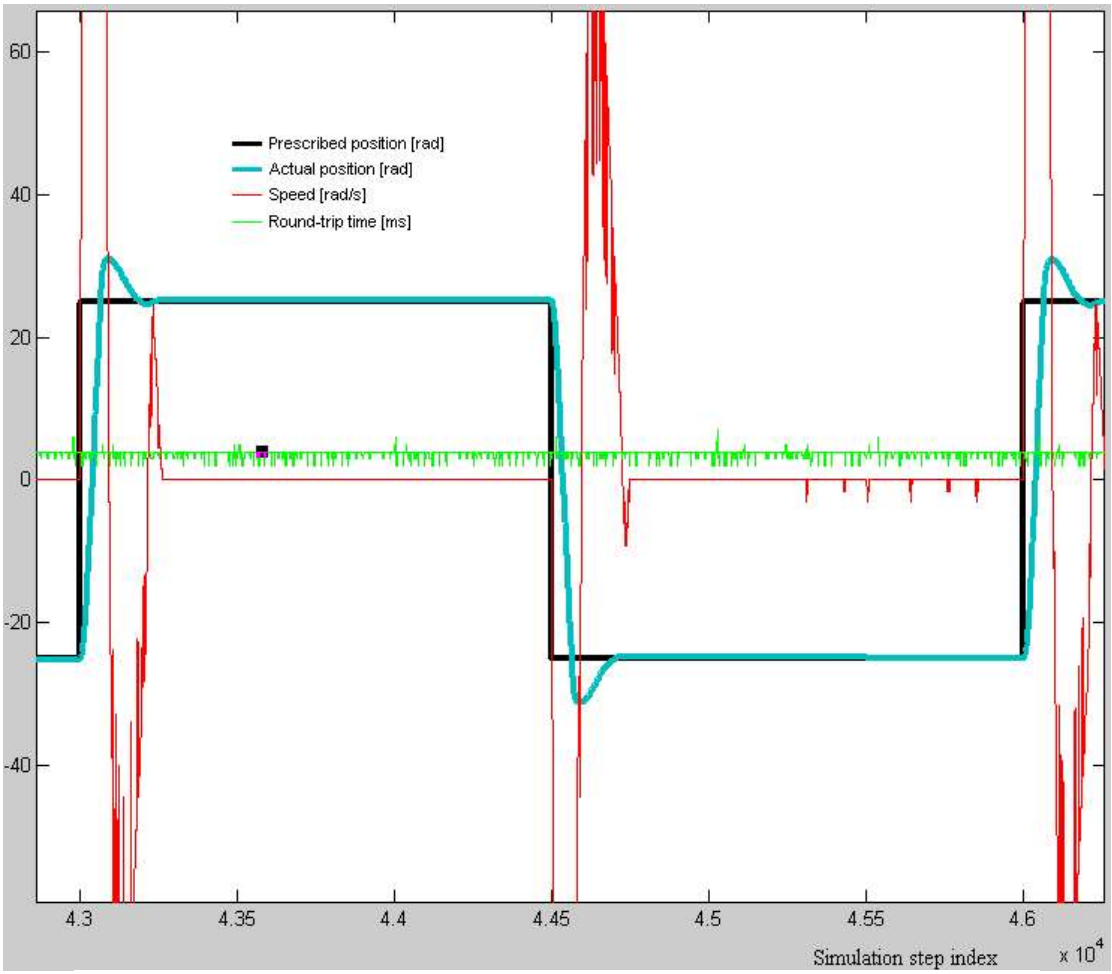

Fig 5.6 PI position control, with a square wave used as prescribed position

The controller is quite quick to react, the time needed by the actual position to reach 90% of the prescribed value, as measured on the square wave, being 120 milliseconds. Overshoot is small (6 radians for a 50radian step) and oscillations quickly damped on the square wave.

Despite the variation in round-trip time however, the controller can do its job of bringing the motor to the desired position, even when the load on the motor varies (for example, by squeezing the shaft to make it harder to turn, and thus increasing the d parameter). While not suited for precise or safety-critical uses, this setup suggests that hobby or educational process control or robotics applications may employ USB connections even without real-time operating systems.

5.3 Linux – libusb

libusb, an open source solution hosted on http://libusb.sourceforge.net, supports various operating systems (Linux, various versions of BSD, MacOS X; a Windows

package may also be available in the future) and is meant as a general platform upon which USB device drivers can be built[5.6]. Well documented, easy to use, it is free of charge. Linux systems do not necessarily come with libusb, in which case an administrator must install it.

Although initially using the HID driver was attempted, this approach was abandoned vendor-class was used instead. An HID would have been claimed by the poorly documented Linux driver. While it is possible to tell it to ignore some devices, this requires a few configuration files to be rewritten and the kernel and driver recompiled.

In the end, because vendor-class imposes no requirements on the devices and libusb can connect to any device not claimed by another driver, this solution proved simpler than the Windows XP HID-based approach.

Finding and connecting to a USB device is similar to the Windows method: all USB devices are cycled through, looking for one that matches a given Vendor Id - Product Id pair. If the device is found, a set configuration request is issued and an interface claimed. Note that all requests necessary to connect to or disconnect from a device must be made explicitly, because no other driver will automatically handle them.

Sending or receiving data must be made, in this case, through usb_interrupt_write and usb_interrupt_read respectively. Other transfer types (bulk, control, isochronous) have their own functions. All of these functions are blocking and should return when the operation is either finished or timed out. Annex “A6: Testing USB on Linux” gives the source code.

All work shown here and all measurements were performed on a Linux system with the scheduler running every millisecond.

5.3.1 Running a USB communication in Matlab (Linux)

A Simulink model (joystick_test_LINUX.mdl) identical to the Windows one except for the MEX function used to connect to the USB, was used to assess Matlab's connection to the USB on Linux.

A plot of measured round-trip times is shown below. Round-trip times are measured by the same method as described in the previous chapter. The device prepares data for the Host on SOF events. The plot shows round-trip time measurements with the model running for about five minutes. A simulation step takes approximately 4 milliseconds.

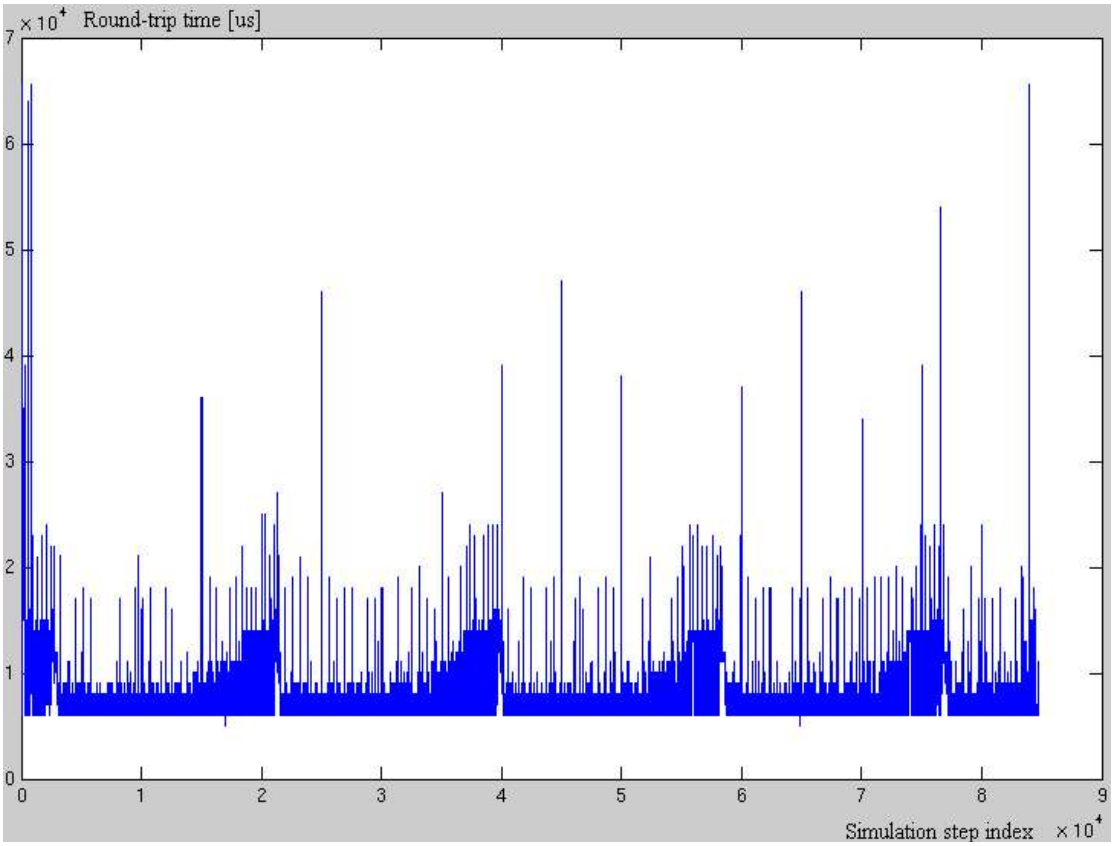

Fig 5.7 Matlab (LINUX) round-trip times, in normal mode

The typical performance is about 6 milliseconds. Random spikes, like in the Windows version, are present. A periodic worsening (about each 66 seconds) is also visible, probably caused by some process in the system. Just like the Windows case, the XY Graph should be closed as it severely affects performance.

Linux Matlab's poor performance (compared to Matlab on Windows) is due to the fact that the functions used here are blocking. This means that the transmission request is only processed once the reception is finished, and never will the two share the same frame, unlike the Windows case. The next chapter will show a way around this problem.

5.3.2 Running a USB communication in joystick demo

The old joystick demonstration program was used for the new, netX based joystick. The application remained unchanged except for the USB communication code. The old driver proved incompatible with the new kernel and had to be replaced with the libusb-based code.

The single difference between the Matlab and the joystick demo communication is the use of threads and signaling in the case of the latter.

The joystick demo is split in three threads: a read, an update scenario, and a write thread. The update scenario thread is called periodically at two milliseconds, an interval which can be achieved reliably if the scheduler works every millisecond and the update scenario thread uses few processor resources. It uses the most recent packet of data available from the read thread. The read thread itself loops continuously, and via a mutex protocol it sends data to update scenario. The write thread waits for a signal from update scenario before attempting to send data on the bus.

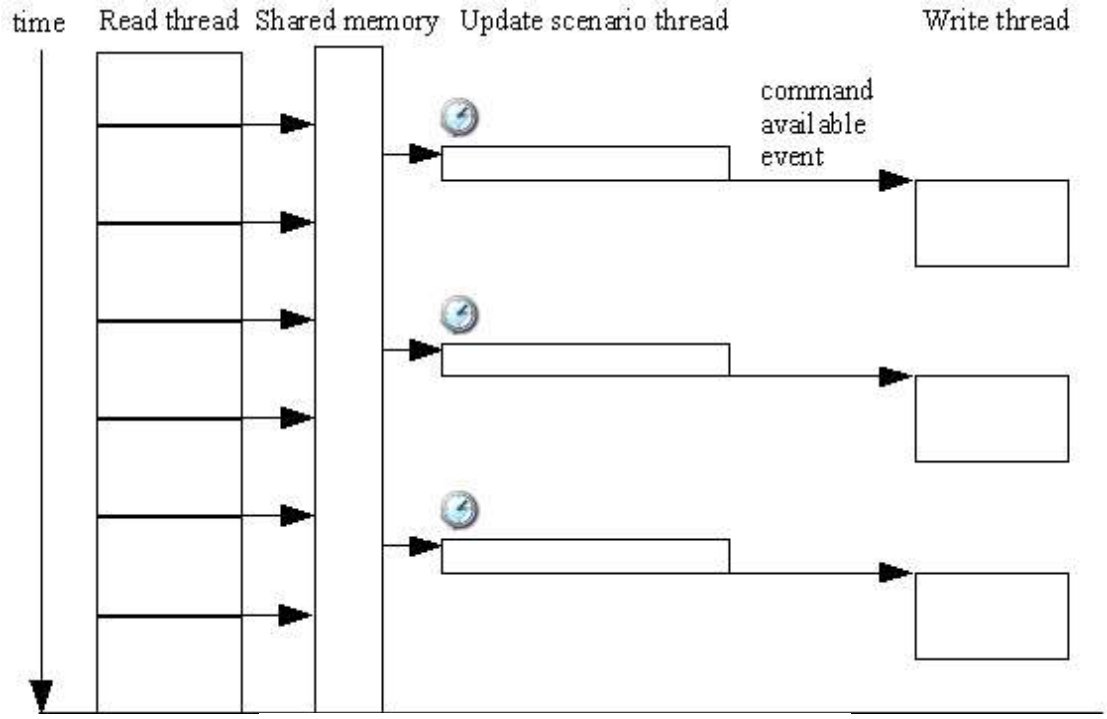

Fig 5.8 Joystick demo thread synchronisation

Round-trip time measurements are sent by the joystick on each communication. The joystick demo application accumulates a sum of them in the update scenario thread. It counts how many update scenario steps happened in the course of three seconds, and will calculate an average round-trip time for those three seconds by dividing the round-trip time sum to the number of update scenario runs. It will also compute the average interval between two runs of update scenario. Once an average round-trip time is calculated, the roud-trip time sum is reset to prepare for the next measurements.

These calculated values (average round-trip and update times in a three second interval) are displayed in the console window of the application. Taking 37 consecutive of these values, the following were obtained:

	Min [us]	***Max[us]***	***Average[us]***	***Standard deviation[us]***
Round-trip	2076	3863	2812.97	394.09
Update	2006	2090	2021.14	18.71

Table 5.1 Joystick demo thread round-trip and update times statistics

The joystick demo application performs much better than Matlab, because of its usage of threads. It still, however, does not perform with hard real-time reliability, round-trip times varying by large amounts. Like the Windows case, this might be enough for some non-demanding applications.

5.4 QNX – usbd

usbd is the library meant to assist in developing USB device drivers for QNX[5.7]. The current version has some limitations (callbacks for vendor requests are not supported, High-speed USB has no support for isochronous or split-isochronous transfers[5.8]) but these did not affect my attempt to connect a device via USB to a QNX computer.

USB is not always automatically started when QNX boots, so the root user should issue an

```
io-usb -duhci -dehci -dohci
```

command to enable the USB ports.

Like in the Linux case, all connected USB devices are enumerated, and when one with the looked after Vendor and Product Ids is found, it is configured and its interface claimed.

Unlike Linux, data transfers are two step and non blocking. First, a USB Request Block (URB) must be prepared, then the URB must be submitted to the USB stack. When the transfer actually takes place on the bus, a notification via callback is issued [5.9].

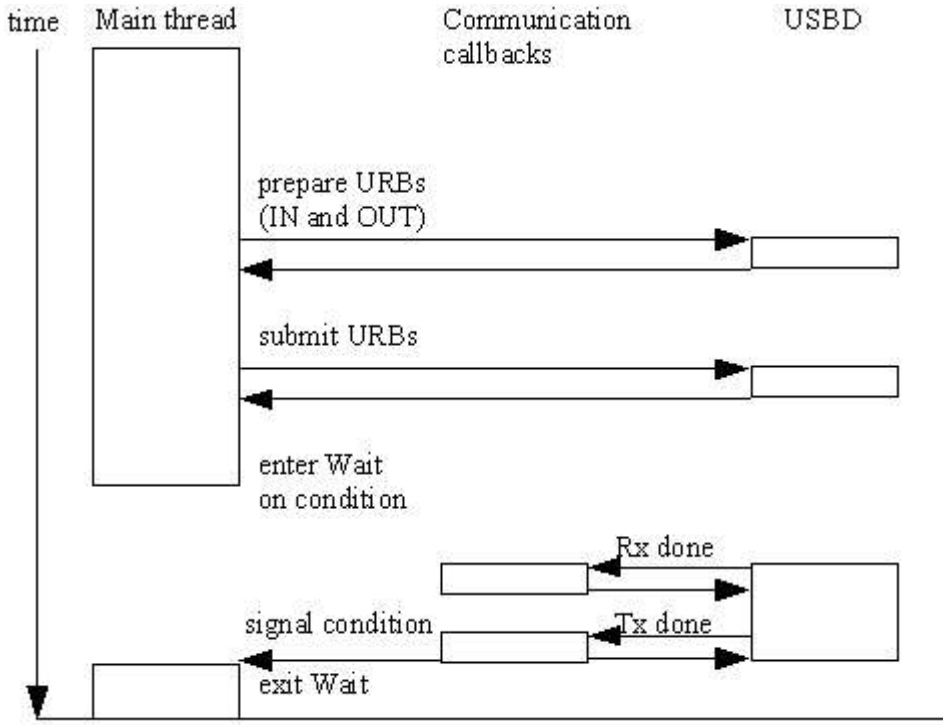

Fig 5.9 QNX: accessing USB with blocking I/O

In order to achieve "blocking I/O" behaviour, that is, have the application wait until the bus operation completes but without loading the CPU in this time, POSIX condition signalling is used. The main thread issues requests to the USB stack, and then enters a wait on condition (it releases the CPU throughout this time). Signaling to fulfill the condition happens in callbacks, activated by usbd when the requested communication events are completed. The order of completion is not important, both events, reception and transmission, must finish before the condition is signalled and the waiting exited.

5.4.1 Running a USB communication on QNX

Unlike the previous operating systems, a USB communication flow on QNX adds a delay between the moment that a USB event takes place, and the moment that the application takes note of it. Specifically, an application is not informed immediately that a request (a transmission or a reception) has been completed. Rather, the application is notified on the start of the next frame. This is done probably because it simplifies the USB driver code, and assists in keeping real-time behaviour.

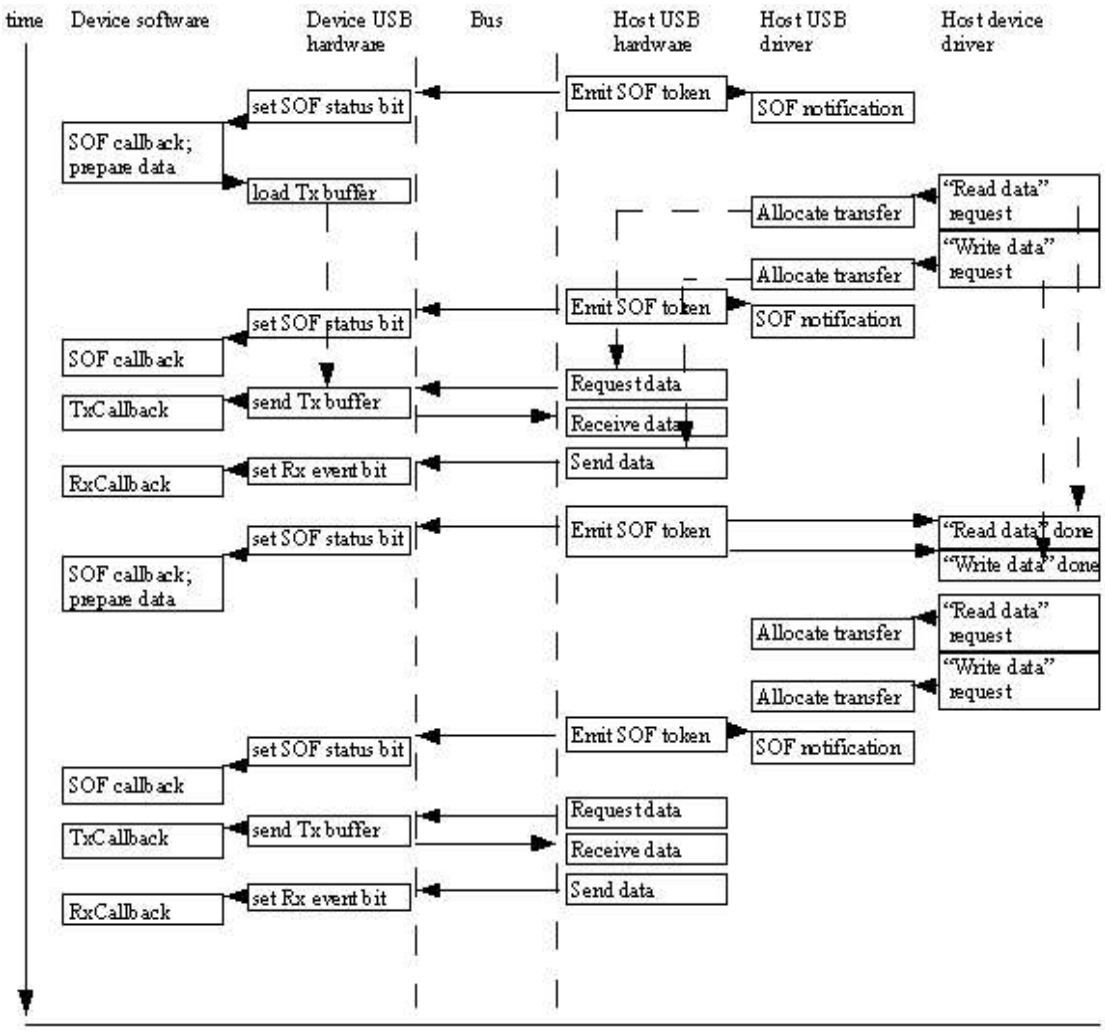

Fig 5.10 USB communication flow on QNX, variant one

In the figure above, an example communication flow between a "regular" device and a QNX Host is shown. The consequences of the extra delay inserted by QNX are that the data sent by the device waits for 2 milliseconds before reaching the application, and the round-trip time becomes 3 milliseconds. An assumption is made that once the application on the Host is notified of new data, it can calculate a response and request new transfers within one millisecond.

On the first SOF the device prepares data, but this data will only be sent on the second frame, and reach the application in the third. The application will prepare a response, which will reach the device in the fourth frame, 3 milliseconds after the initial data was sent. (The data sent to the device on the second frame cannot be a response of the application to data it has not received yet; the data for that transfer has been prepared and copied to the USB driver buffer before, when the transfer was allocated).

The second packet of data that the device prepares, on frame three (because on the second SOF the first transmission is not yet ready), is only sent to the Host on frame four, and will be read by the application on frame five. The reply to this second packet

will reach the device on frame six. Communication continues in the same fashion over the next frames.

There is nothing, short of tampering with the USB driver, that can be done, on the Host side, to correct this. On the device however, there is a solution to attempt to reduce the round-trip time slightly. The idea is that the device should only prepare data on a SOF that will be followed by transfer requests, an "active SOF". The problem then is for the device to decide which SOFs are "active" and which are not.

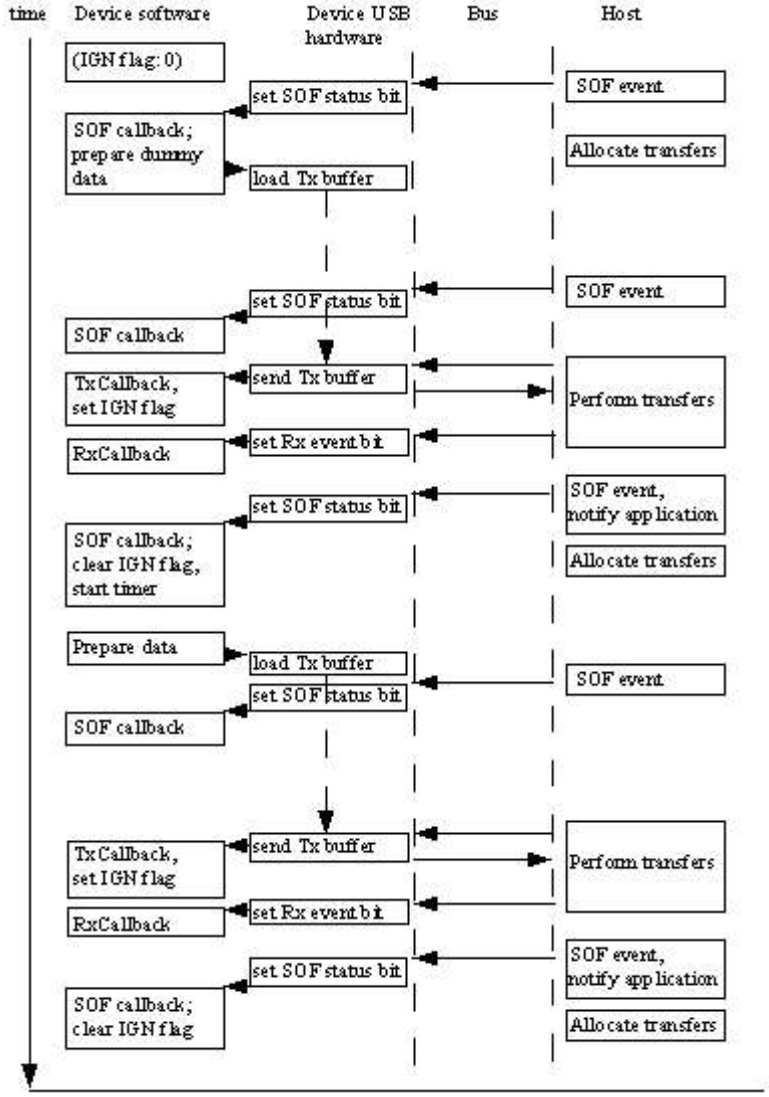

Fig 5.11 USB communication flow on QNX, variant two

Figure 5.12 shows a simple protocol that the device can follow to decide whether to prepare data or not. On first SOF, the device prepares some dummy data to make sure requests from the application on the Host are successful. The data will, just like before, be sent in the second frame. However, the device will not prepare data on the third SOF, because it knows now that it will be "not active" (its frame will not contain transfers). Instead, since the pattern of active/inactive frames is predictable, the device can tell just by counting SOFs which frames will have transfers and which will not. It

will then try to prepare data as close as possible to the start of an active frame. When an inactive frame starts and the device receives the SOF packet, it will start a timer which will elapse slightly before the start of the next frame. When this timer elapses, the device prepares data to be sent to the Host.

The interval between the timer elapsing and the start of the next frame is called in this document the “anticipation interval”. Its purpose is to make sure that when the active frame starts, the device has the data prepared. The anticipation interval only needs to be long enough to give the device time enough to prepare the data.

The round-trip will only be 3ms on the first transfer after establishing a connection; in this transfer the device finds out which frames are active. After that, using active frame anticipation, the round-trip can be reduced to 2ms (plus the anticipation interval).

This protocol needs two assumptions in order to deliver the two-millisecond round-trip time: the application on the Host can calculate a response within a millisecond of being notified of events on the bus, and will prepare both a transmission and reception every two milliseconds. If these assumptions do not hold, it should still be the case that a regular pattern of active/inactive frames will occur, and the “active frame anticipation” can be adjusted accordingly.

To measure the round-trip time, a simulink model (joystick_test_QNX.mdl), compiled with RT Lab, was run on a QNX computer with the graphical interface provided by a Linux computer running Simulink in external mode.

As expected, QNX performs reliably and can maintain a round-trip time with very small variations. A plot is given below.

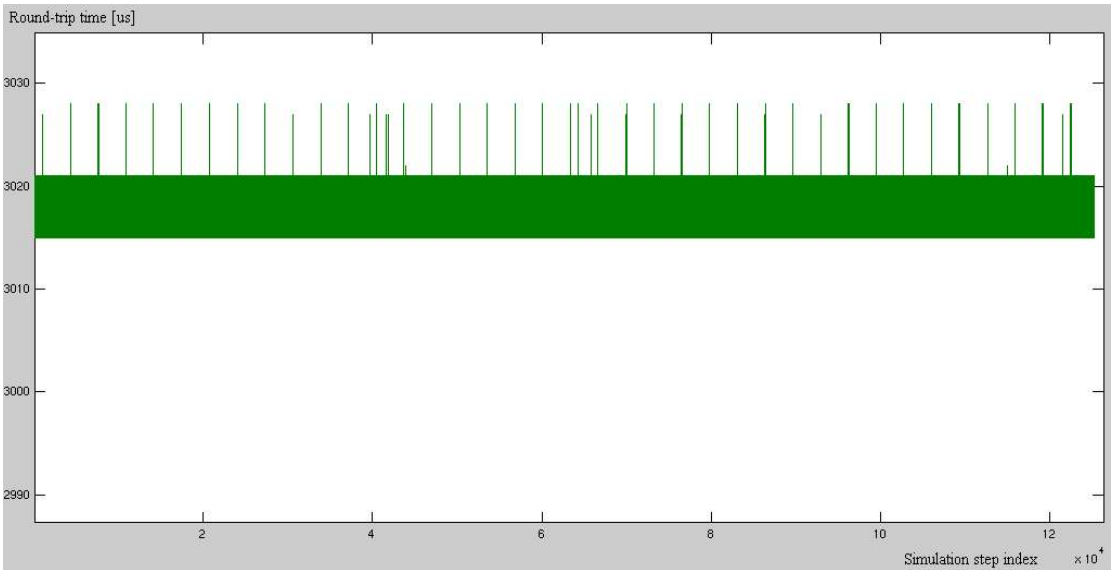

Fig 5.12 Round-trip time on QNX

The round-trip time stays at about 3000 microseconds, with very small differences between one value and the average. This remains so even if the Simulink model is stopped and then restarted.

The screenshot below shows a fragment of event tracing for the QNX system. The USB interrupt is generated every 1.995 milliseconds as measured by the QNX clock. This proves that notifications arrive to the application only every two milliseconds.

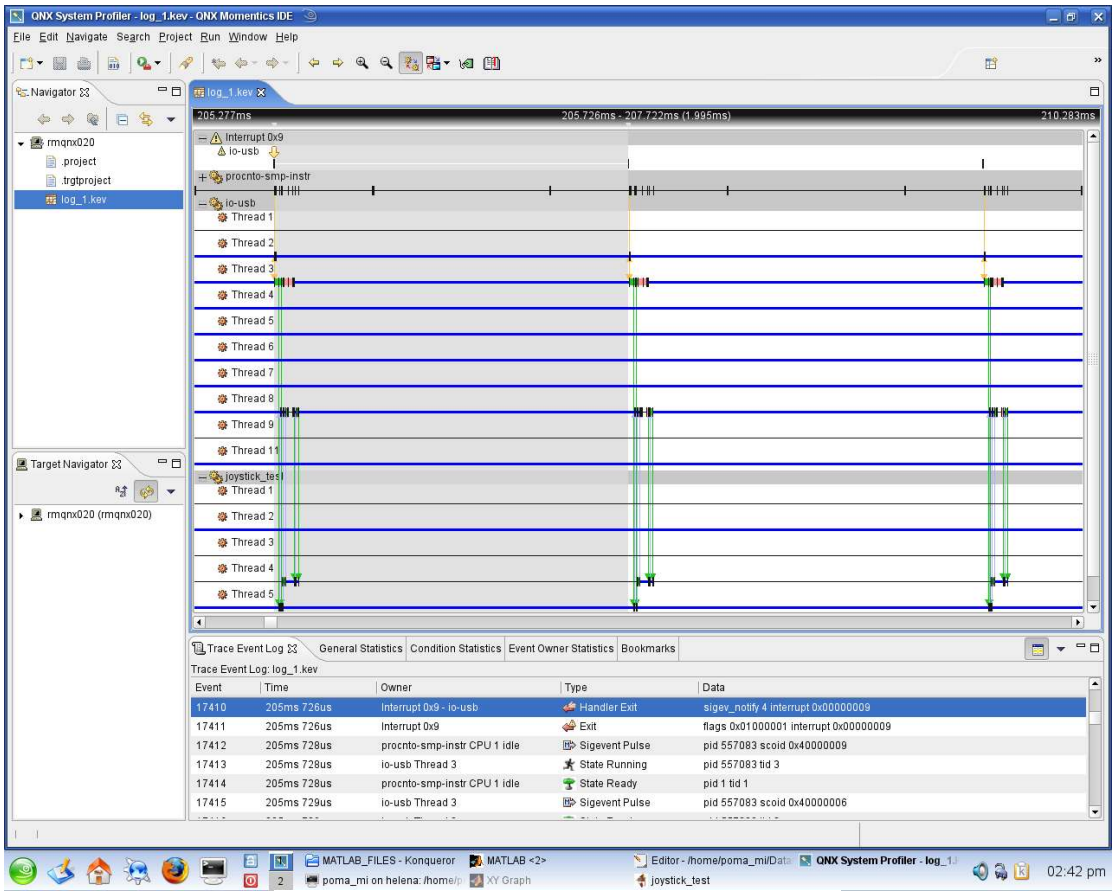

Fig 5.14 Events tracing in QNX Momentics

The screenshot also shows the calls to condition wait and condition signal that produce the blocking I/O behaviour.

Having the device anticipate the active SOFs can reduce the round-trip time to 2 milliseconds plus an anticipation interval. The plot below shows a communication using 400µs anticipation, running for 6 minutes.

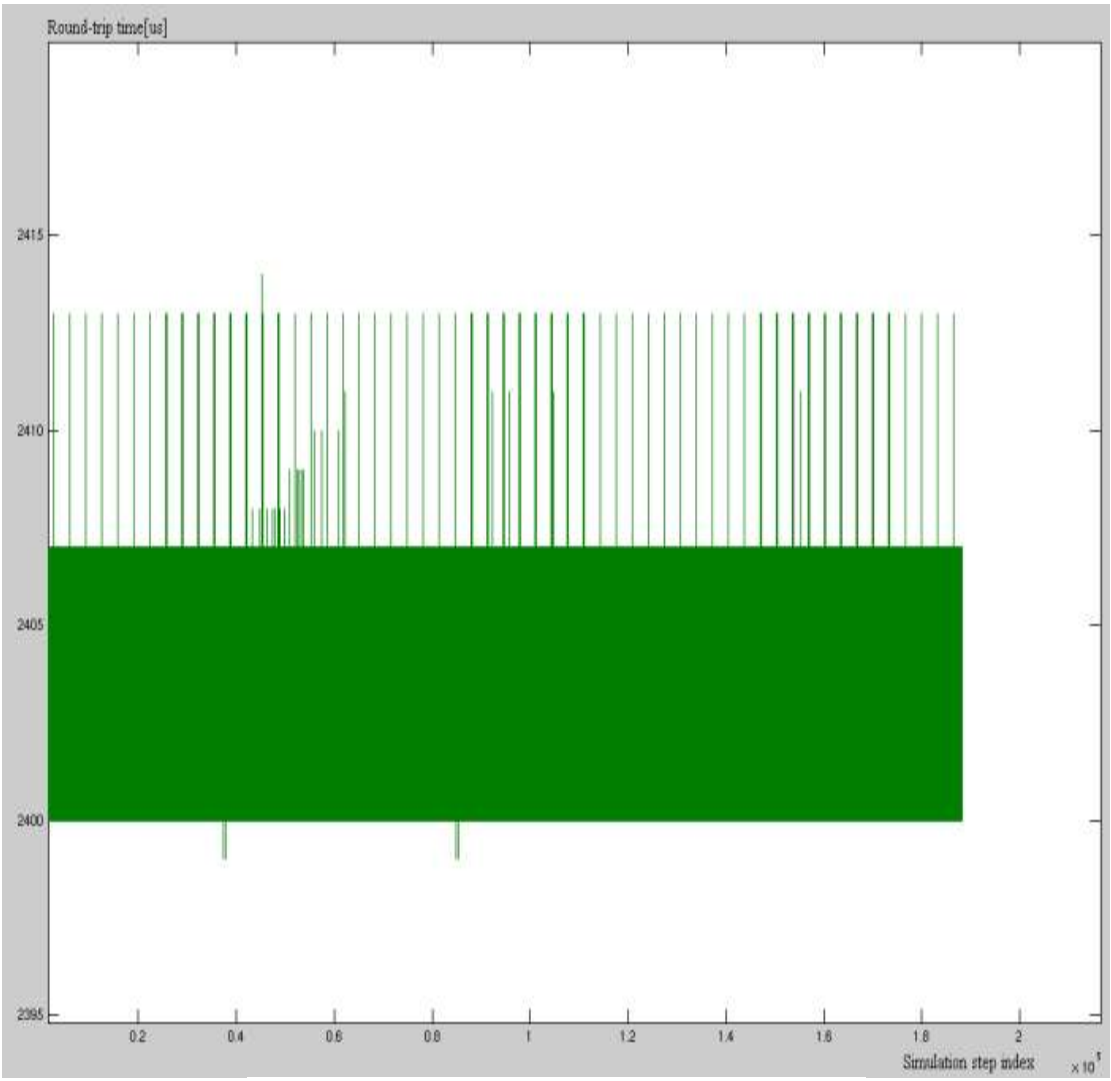

Fig 5.15 Round-trip time with SOF anticipation

The anticipation interval can be much smaller however, depending on how much the device needs to prepare data. This usually means a few tens of microseconds.

Fig 5.15 Round-trip time with SOF anticipation

6 Conclusions

During this diploma thesis,a USB firmware library was developped for the netX processor. It is reconfigurable and portable, useful on different projects and hardware platforms. On the PC-side, several small pieces of code were implemented, in order to allow a USB communication to be opened and sustained. The USB communication code in the ROKVISS joystick demo has also been changed, because the old driver was not workable with the new kernel. Measurements of round-trip time for the USB connection, on various operating systems, have been performed.

These measurements show that the round-trip time under Windows or Linux is variable, highly vulnerable to various loads on the system. These loads cannot be controlled by an application, and can make the system cease USB communication for as long as or longer than 65 milliseconds. However, such instances are rare and round-trip times tend to cluster between 2 and 5 milliseconds. Some applications (e.g. educational or hobby robots) may tolerate this variation, especially if fitted with safety breaking features in case communication ceases for longer periods of time. Neither Linux nor Windows are real-time operating systems, so these results are not surprising.

QNX can maintain a round-trip time of 3 milliseconds (and a sample time of 2 milliseconds) on a USB connection. Using anticipation of active frames, the round-trip time can be reduced to 2ms plus the anticipation interval.

The total amount of data that can be transferred (either in or out) between a computer and a device, on full-speed USB, using interrupt transfers, is 1920 bytes maximum if the device supports 30 data endpoints. With a sample time of 2, and round-trip of 3 milliseconds, this is adequate for controlling a robot arm, and leaves a fair amount of time for data processing.

For best performance, overlapped I/O or separate read and write threads should be used. Also, only one device should be connected to a USB port if time performance is critical, because this will reduce jitter caused by the variable transfer order on the USB. Since each USB port on a computer has a bus of its own, independent of other ports (as the Windows control pannel or Linux lsusb command show), and since there are several USB ports on a computer, this is not a severe issue.

As a future development, High-speed USB should be tested. A newer standard, it is for now less available on microcontrollers, but promises potential round-trip times in the hundreds-of-microseconds range. The amount of data that can be carried by a tranfer also increases (a single interrupt endpoint can carry 1024 bytes), meaning that round-trip time will be determined more by speed of data processing than by bus characteristics.

7 Annexes

7.1 A1: CD contents

/Embedded

/Bootwizard : auxiliary application from Hilscher

/USB_NetX_Joystick : embedded joystick program

/USB_netX_poscont : program for a simple positional control application

/netX_Flasher : auxiliary application for flashing the netX board

/PC

/Linux

/Matlab : Simulink model and MEX function to test the joystick

/Joystick_demo : joystick demo application

/QNX

/Matlab : Simulink model and MEX function to test the joystick

/Windows

/Matlab : Simulink models and MEX functions to test the joystick; a custom mexopts file.

7.2 A2: Joystick program

Located on the diploma CD in the folder /Embedded/USB_NetX_Joystick is the Hitop project “netX_joystick.htp”. This should be built with a Hitop tool-chain. The project file stores file paths as absolute, and may need to be manually edited to correct the paths.

A debug connection via a Tantino probe must be active when attempting to open the project. Through the debugger it can also be downloaded to the netX chip.

The joystick hardware should be available in order to try this application.

In order to try the Anticipated Active SOF communication for QNX, make sure that the __QNX_HOST__ macro is defined in the main.c file. Undefine or comment this macro if you do not wish to use this functionality or do not wish to connect to a QNX

Host.

7.3 A3: Monitoring a motor with the netX

Located on the diploma CD in the folder /Embedded/USB_netX_poscont is the Hitop project “netX_poscont.htp”. Opening,debugging and building are identical to the netX joystick application.

In order to try this application, a DC servomotor with power electronics should be connected to the control board of the joystick. The motor should either be free of load, or have a load such that it can turn several laps. The motors mounted in the joystick do NOT qualify for this.

A quadrature encoder should be fixed to the motor shaft, and its data and supply lines also brought to the joystick control board.

7.4 A4: Flasher App

Located on the diploma CD in the folder /Embedded/netX_Flahser is the Hitop project “netX_Flasher.htp”. Opening,debugging and building are identical to the netX joystick application.

Flashing an application to the joystick control board is not done directly from Hitop, because it does not support the kind of Flash chip on the control board. Instead, follow these steps to Flash an application:

- build the application that you wish to flash
- run the Bootwizard program from Hitop on the elf file resulting from the build. Run the “Build bootimage” wizard, selecting “Internal RAM” as the destination and “MW209B 8bit Parallel FLASH” as the source devices. The Bootwizard application, with an updated xml file, can be found on the diploma CD in the / Embedded/Bootwizard folder
- the Bootwizard will produce a bin file from the elf, adding a bootimage. Use the BinToC application in the netX_Flasher folder to convert this into a c-file. A batch file is provided as an example of how to do this.
- Compile the resulting c-file within the flasher application (replace the CDump.c file with the new c-file). Link, load and run the

resulting application.

The flasher app simply uses a c-file to store a large array of bytes (the program that needs to be flashed) and copies it to the Flash memory chip on the joystick control board.

The Bootwizard will make an application built by Hitop into an application that can be booted on the netX. It prepends a bootimage, giving some information about the application to the netX's hard-coded bootloader.

7.5 A5: Testing USB on Windows

7.5.1 A5.1: Joystick_test for Windows

The Simulink file joystick_test.mdl is necessary for this purpose. It can be found on the diploma work CD, /PC/Windows/Matlab folder.

Before this model can be used, the readROK.cpp file must be built by issuing a

```
mex readROK.cpp
```

command in the Matlab window. The Windows DDK must be installed on the computer for the command to succeed, and the mexopts file must be adjusted by adding include paths for the HID headers from the DDK, and the libraries hidsdi.lib and setupapi.lib. A sample mexopts file is also included on the diploma CD.

7.5.2 A5.2: Motor position control in Simulink

The Simulink file motcontrol.mdl is necessary for this purpose. It can be found on the diploma work CD, /PC/Windows/Matlab folder.

Before this model can be used, the motINTF.cpp file must be built by issuing a

```
mex motINTF.cpp
```

command in the Matlab window. Just like the previous case, the Windows DDK is necessary for the command to be successful, and the mexopts file for Matlab must be adjusted accordingly.

The control parameters must be assigned values before the model can start. This can either be done by manually assigning values to them in the Matlab window like this

```
ki_pi = 0.120;
```

```
kp_pi = 0.110;
```

or by following the procedure described in the next section.

A5.2.1: Tuning the PI controller

The Bode_pos_PI.m file contains a function to tune the control parameters. Invoke the function by issuing the following command

```
[kp_pi, ki_pi] = Bode_pos_PI(J, d, omega_g, Ts, phase_margin)
```

where J, d, omega_g, Ts and phase margin are the various parameters (inertial moment [A*s^2], viscous friction[A*s], omega_g [rad/s], Ts [s], phase_margin[deg]).

The function will return a [0, 0] pair with the message "Obtaining phase margin impossible" if the phase margin is too large for the selected omega_g, given the other system parameters. If however the phase margin can be obtained, a pair of values is returned and bode plots of gain and phase for the open-loop system are produced to check the results.

kp_pi and ki_pi are authomatically written to the Simulink model.

7.6 A6: Testing USB on Linux

7.6.1 A6.1: Joystick test for Linux

The joystick_test.mdl file from the diploma CD, /PC/Linux/Matlab folder, can be used to test communication with the netX joystick. It contains a MEX function that must be built by invoking in the Matlab window the command

```
mex readROK.cpp -lusb
```

In order for the command to succeed, libusb must be installed on the computer.

7.6.2 A6.2: Joystick demo Linux

Located in the /PC/Linux/Joystick_demo on the diploma CD, it can provide a full test of the joystick functionality: USB communication as well as force-feedback. This folder contains a KDevelop project, as well as a prebuilt joystick application.

To run the demo, type the name of the executable (joystick") in the Linux console. The joystick itself should be connected to the PC before the application is started.

Once communication between the application and the joystick is established, move the joystick handle to the lower-left corner of its work-space and press “l”. Then, move the handle to the upper-right corner and press “p”. The position output from the joystick is now calibrated to move thorughout the entire demo window. After this, the demonstration can be used to show the force-feedback behaviour of the joystick.

In order to be built, requires libusb to be installed on the computer. For best performance, the Linux kernel should be recompiled so that the scheduler runs at 1kHz frequency.

7.7 A7 Testing USB on QNX

The Simulink model joystick_test_QNX.mdl, located in the /PC/QNX/Matlab folder, can be used to test USB communication with QNX. The procedure is slightly different than the other systems, in that no mex command is issued. Instead, a Real-time Workshop target must be installed by issuing a

```
mex readROK.cpp -lusb
```

command in the Matlab window. After this, the model should be built by using the Real-Time Workshop build command.

Three files are generated: joystick_test_QNX, joystick_test_QNX_start.sh, joystick_test_QNX_stop.sh. In order to make these files executable from a QNX computer, these commands must be issued in a Linux console:

```
chmod 755 joystick_test_QNX_start.sh
chmod 755 joystick_test_QNX_stop.sh
chmod 755 joystick_test_QNX
```

After this, the model can be started on the QNX computer by issuing

```
./joystick_test_QNX_start.sh
```

On the Simulink side, the “Connect to real-time target” option should be selected, and the model can now be interfaced from Simulink. Use the “Stop real-time code” to stop the model. This will finish it on both QNX and Simulink sides.

References

2.1: “Universal Serial Bus Specification, revision 2.0”, April 27, 2000, section “4.4 Bus Protocol”

2.2: id. , section “4.1.1 Bus Topology”

2.3: id. , section “4.2 Physical Interface”

2.4: id. , section “5.3.3 Frames and Microframes”

2.5: id. , section “4.6 System Configuration”

2.6: id. , section “9.6 Standard USB Descriptor Definitions”

2.7: id. , section “9.4 Standard Device Requests”

2.8: id. , section “9.7 Device Class Definitions”

2.9: id. , section “5.11 Bus Access for Transfers”, subsection “5.11.1.2 USB Driver”

2.10: id. , section “5.3.1 Device Endpoints”

2.11: id. , section “9.6.6 Endpoint [Descriptor]”

2.12: id. , section “5.5 Control Transfers”

2.13: id. , section “5.8 Bulk Transfers”

2.14: id. , section “5.7 Interrupt Transfers”

2.15: id. , section “5.6 Isochronous Transfers”

2.16: id. , section “8.4.5 Handshake Packets”

2.17: id. , section “7.1.12 Frame Interval”

2.18: id. , section “5.6.4 Isochronous Transfer Bus Access Constraints”

2.19: http://www.usb.org/developers/defined_class

2.20: http://www.usb.org/developers/defined_class/#BaseClassFFh

4.1: G. Hirzinger et al., “ROKVISS – Robotics Component Verification on ISS”

4.2: Klaus Joehl et al., “High Fidelity USB Force Feedback Joystick”

4.3: “netX Product Brief”, April 20 2007

4.4: “EtherCAT Communication Specification, version 1.00”, section “4.1 Operating principle”

4.5: Matthias Faehse, “Entwicklung einer Steuerelektronik fur den DLR kraftreflektierenden Joystick”, February 28, 2007

4.6: “netX Program Reference Guide”, April 20, 2007, section “6.6 System time with IEEE 1588 functionality”

4.7: id., section “6.9 USB- Serial USB-Interface”

4.8: (netX forums:) http://board.hilscher.com/viewtopic.php?t=176

5.1: The http://www.lvr.com/hidpage.htm

5.2: “Device Class Definition for Human Interface Devices, version 1.11”, June 27, 2001, section “2.1 Scope”

5.3: id. , section “4.4 Interfaces”

5.4: “Device Class Definition for Physical Interface Devices, version 1.0”, September 8, 1999, section “2 Functional Overview”

5.5: Ivan Bogdanov, “Conducerea cu calculatorul a actionarilor electrice”, Ed. Orizonturi Universitare, 2004, section “4.1.3.1 Modelul matematic operational al MCC”

5.6: http://libusb.sourceforge.net/documentation.html

5.7: http://www.qnx.com/developers/docs/6.3.0SP3/ddk_en/usb/about.html

5.7: http://www.qnx.com/developers/docs/6.3.0SP3/ddk_en/usb/beforeyoubegin.html#LIMITS

5.8: http://www.qnx.com/developers/docs/6.3.0SP3/ddk_en/usb/usbd_io.html