COPY 1

USB Implementation to communicate with a Nintendo Game Boy Advance

Author: **Robert Meerman** (0219795)
Supervisor: **Roger Packwood**
Year of Study: **2004-5**

Abstract

This project aimed to enable a Nintendo Game Boy Advance (GBA) to be used as a personal digital assistant by creating a USB communication interface between a Windows PC and a GBA, which would require no hardware-specific drivers or user-configuration.

USB v1.1 is implemented on a Microchip PIC16F877 micro-controller (using the CCS compiler add-on) and a Philips Semiconductor PDIUSBD12 low-level USB chip which handles interactions with the USB bus.

An interface with GBA is achieved via UART and then use the USB implementation by means of the (USB-IF defined) Human Interface Device (HID) class to transfer user-data.

Keywords

USB, Embedded System, PIC, Game Boy Advance, Hardware, PDIUSBD12, PIC16F877

Table of Contents

Table of Illustrations

1 - Introduction

This project was conceived to bring the connectivity of a Personal Digital Assistant to a Nintendo Game Boy Advance portable games console (herein GBA) by creating a USB link between GBA and PC.

This project idea occurred to me when I was considering purchasing a PDA for myself, and realised that the functionality I considered important was actually quite basic (Scheduler, To-do list and Address Book). I personally own a GBA which I usually carry with me and use to pass idle time at bus-stops and so on. One day I thought “If I was to get a PDA, I'd have to carry two things around with me”; I would keep my GBA with me because the games available on a PDA pale in comparison to those on the GBA and would never hold my interest. This, coupled with my knowledge of the GBA “homebrew” community (more detail on page 2) led me to settling on this project.

The main focus of the project was to create a USB implementation that allowed data to be sent between the GBA and the PC, using tools available to the university. A USB implementation *requires* a micro-controller to handle protocol requests, as well as appropriate circuitry to interface with a USB bus, so this project concentrates on creating an embedded system to realise such an implementation.

A variety of USB helper chips are available, usually intended for handling interactions with the USB bus, such as signalling and error-recovery. Many such chips were considered before settling on the Philips PDIUSBD12, chosen because detailed implementation schematics and source code on how to interface with a Microchip PIC micro-controller was available in C (most other example source code was only available in assembly), and the Computer Science department have equipment, expertise and experience using PIC micro-controllers.

The largest hurdle in the project was implementing the necessary protocol functions to enable a USB device to be configured by a host (a PC running Windows XP in this case), a process termed *enumeration*. Until the device is configured by the host, there are no user-accessible events or data to aid debugging at the host, rendering software “USB sniffers” useless for this crucial portion of the development. Debugging had to be done at the device, but interleaving statements which transmitted this debugging data complicated development and required careful progress.

Once the device did enumerate successfully, an implementation of the Human Interface Device (HID) class, as defined in a USB class standard, was implemented and used to send user data between the PC and firmware, while the communications between the developed hardware and GBA were handled by a UART link.

1.1 - Introduction to Game Boy

The Game Boy series are battery powered hand-held games consoles sold by Nintendo. The first in the series, the Game Boy, was released in 1989 and became highly successful, selling 32 million units in its first 3 years[1].

The original Game Boy system was based on an 8-bit Z80 CPU running at ~4MHz, with 8kbit of internal RAM and a 4-shade grey-scale display. Many different variations have been produced since the original release (*Game Boy Play It Loud!*, *Game Boy Pocket*, *Game Boy Light and Game Boy Color*) with little change to the system's capabilities, while growing in popularity as more games became available.

Illustration 1.1.1 - A Game Boy Advance

In 2001 the Game Boy Advance was released, and offered a considerable increase in power and capabilities compared to its predecessors. This new system was based on a custom 32-bit ARM processor running at ~17MHz with 256kBytes of RAM, had a colour display capable of displaying 15-bit colour, and was backwards compatible with the old system (allowing all old games to be played on it), which did much to help it become successful.

Soon a variation of the new system was released: the Game Boy Advance Special (GBA:SP) – aside from some cosmetic changes (clam shell layout, front-lit[1] screen and internal lithium-ion battery) the system is identical to it's predecessor, the "classic" Game Boy Advance.


Illustration 1.1.2 - A Game Boy Advance SP (GBA:SP)

The 32-bit ARM processor the GBA/GBA:SP is based on is much more compatible with the C programming language that the 8-bit Z80 processor used in the previous Game Boy systems, and so when equipment became readily available for moving user code onto GBA hardware in early 2002, an active internet "home-brew" software development community[2] sprung up, and soon

1 As the GBA:SP follows the GBA it uses the same reflective LCD component, so a back-light was not possible.
2 Example community sites include: `www.gbadev.org`, `www.devrs.com/gba`, `www.ngine.de` and

produced the necessary tool-chains and technical documentation to enable any hobbyist to write and compile C code which would run on the GBA hardware.

There are two main methods to getting home-brew code to run on a GBA – writing the compiled binary onto a flash cartridge which is compatible with system, or making use of the GBA's bootstrap loader.

A variety of flash cartridges designed specifically for this purpose are commercially available, and are written using either a custom hardware writer, or much more commonly in recent times, by making use of the GBA's bootstrap loader.

When there is no cartridge in the GBA (or the user holds down certain buttons when powering the unit on) the system enters multiboot mode (sometimes called *netboot*), where it listens on its communications port for a host that wishes to send bootstrap code. Intended to allow multiple units to play a multi-player game when only one game cartridge is available. Bootstrap code is limited by the size of available RAM: 256kB, however this is sufficient for most home-brew applications. This mode is often used to transmit a small “loader” into the GBA which then acts as a pass-through between PC and GBA, allowing appropriate PC software to rewrite a compatible flash cartridge inserted into the GBA via the GBA hardware – this method is preferred by most as it reduces wear on the cartridge's tracks (as there is no need to keep removing and inserting the cartridge between GBA and a cartridge writer) and is both cheaper and more compact than buying a special writer.

The GBA does not offer any operating system, and it is up the developer to make appropriate use of the available hardware – for instance instead of drawing moving objects on the display pixel-by-pixel every frame of animation, it is much better to draw them once into Object Memory and then place one of the display layers into sprite-mode and set the Object Attribute Memory appropriately for each object – setting coordinates, rotation and opacity – thereby allowing the GBA's video hardware to take care of rendering the object while the CPU handles other tasks.

The communications port of GBA is intended for creating small wired networks of 2-4 GBA units for multi-player games and for interfacing with peripherals. So apart from supporting propriety Nintendo networking protocols it offers a general purpose mode and a buffered UART mode. The general purpose mode allows individual pins to be set logic-high, logic-low or as input while the UART mode is suitable for interfacing with RS232 once some voltage-level conversion has taken place, when used the GBA offers hardware flow control using Clear-To-Send/Request-To-Send (CTS/RTS) lines.

www.thepernproject.com

1.2 - Introduction to Universal Serial Bus

Universal Serial Bus is a serial bus standard for connecting devices usually used to connect peripherals to a PC, but it is not limited to this and has been used in set-top boxes, games consoles and PDAs.

Illustration 1.2.1 - A USB plug

USB was designed to replace and improve upon legacy ports (such as RS232 and the IBM parallel port). These improvements include offering devices the option to be powered by the host (with a maximum power consumption of 500mA per physical connector), plug-and-play support, ability to be hot swappable and the ability to instantly add more USB ports by attaching a USB hub (up to a maximum of 127 ports including all hubs, with the root hub (so-called the host) also taking up a port).

Version 1.0 was released in January 1996, and specified a host-orientated design where a single host per bus controls all communication[2]. All bus traffic is initiated and controlled by the host, with the direction being one of IN or OUT and always from the perspective of the host, regardless what context is being discussed – this is the terminology the standard utilises.

When a USB device is attached to a host a process termed *enumeration* occurs where the host queries the device about its capabilities and requirements and then issues commands to place the device in a configured state, while also taking care of loading drivers in the host operating system as appropriate (this includes querying the user for compatible drivers).

This process makes USB truly plug-and-play as the enumeration process is a well-defined part of the standard, allowing devices to be uniquely identified by the host. USB is also fully “hot swappable”, allowing the removal of any device at any time without adversely affecting the operating system[3]. When a device is removed the hub it was connected to makes a note of this event and sets a flag which the host will retrieve when it next polls the hub – device attachments are handled in a similar manner.

A major strength of USB is that it is easy to gain more ports – unlike legacy ports such as the parallel port it is not necessary to open the PC case and insert a PCI / ISA card while the machine is powered off, instead a USB hub can be inserted into a free port while the system is powered to expand the number of USB ports, this is possible because USB hubs are devices themselves. Some products have hubs in them as a secondary function, such as a USB keyboard which provides convenient USB sockets to attach a mouse or PDA/smart-phone cradle. Such a product would

3 This does not imply that no data will be lost or no corruption incurred, only that the system remains operational.

enumerate as a composite device made up of keyboard and hub, and each component would in turn be enumerated.

Unlike legacy ports, USB does not make any assumptions about the data it will be used to communicate. For instance the parallel port was designed with printers in mind, and so of the 25-pins on the PC connector, only 8 are intended for true data while the others are intended to be used as status lines, similarly the serial port was intended for interfacing with terminals and modem equipment, and features a *Ring Indicator* status line on the connector.

Instead, USB connectors only have 4 lines *(V_{BUS}, Data+, Data- and Ground)* and provides the facility to to create logical channels (termed *pipes*) between software *endpoints* on the host/device. These pipes are synonymous to byte streams, while the endpoints are mono-directional addresses. By making endpoints mono-directional the task of implementing the protocol in software (the *firmware*) is simplified as each endpoint direction is implied. This functionality also provides a convenient way to separate protocol data (such as that used in enumeration) from the true data the device sources/sinks.

USB was intended to be a universal replacement, and to that end it support 4 transfer types:

- *control transfers* – used to issue requests specified by the USB standard, for instance those used in enumeration conversations; however it can also be used for arbitrary data. All devices must support control transfers or they are unable to enumerate.
- *isochronous transfers* – data is transferred at a guaranteed rate (usually as fast as possible), this is the only transfer type where data is not retransmitted on an error. Example use: real-time audio/video
- *interrupt transfers* – have guaranteed latency (to an upper bound), Example use: pointing devices and keyboards.
- *bulk transfers* – large sporadic transfers which make use of all available bandwidth without guarantee of rate of latency. Lowest priority transfer type. Example use: file transfers.

USB Classes

Devices often share attributes in common, for instance all mice send movement data and button clicks, and so it is useful to define a standard which encompasses these common attributes. USB has a notion of “classes” which perform just such a purpose, and has defined a number of classes, such as Human Interface Device (HID) class (which I have used in this project) and mass storage device class.

Classes are like templates on which implementations can be created, their implementation can

extend the class, so long a it doesn't contradict the class specification. This is useful in, for instance, USB flash drives with special functionality, perhaps an encrypted area of memory – when attached to the system the host may not be able to find specific drivers for the device which can perform the secure transactions, and so loads the operating system's default mass storage class driver instead; making the basic functionality available to the user until a custom driver is installed (at which time the device will be re-enumerated and the special driver will be loaded).

As stated previously, this project made use of the Human Interface Device (HID) class, which uses a standardised structure, called a *report*, to describe data being transmitted in either direction, and allows multiple report structures to be defined on an endpoint, enabling efficient use of the system's available bandwidth or power – important in laptop environments, especially if the device is bus-powered by the host – by allowing the report to be chosen dynamically by the device. Useful in monitoring applications as a short “no change” can be sent, or a group of related readings can be sent when something changes.

The report structure defined in the standard even has the option to attach associated compositions of SI units[4], such as grammes or seconds, to any data in the report, potentially allowing a generic “monitor” application on the host to parse data from almost any HID-class device and display readings in a (potentially) useful manner.

USB Speed & Traffic

USB v2.0 supports 3 speeds, termed: low-speed, full-speed and high-speed, supporting a *maximum* throughput of 800 B/s, 1.216 MB/s and 53.248 MB/s respectively, if the bus is otherwise idle and that bulk transfer mode is used.

Low and Full speed were part of the USB v1.0 specification, while High-speed was added in v2.0. This project made use of a full-speed USB interface chip, but not of USB's bulk transfer mode, using interrupt and control transfers only.

All USB traffic is controlled by the host, and the convention in the USB specification is to discuss traffic from the host's perspective; all traffic is denoted to be directed IN or OUT. 'Control IN' therefore denotes a control transfer from device to host, and a 'Control OUT' to be from host to device, even when in the context of writing device firmware.

Despite the name *interrupt*, interrupt transfers do *not* interrupt bus activities, in fact they cannot initiate a connection to the host at all. This is because, as stated above, *all* traffic is controlled by the host – interrupt endpoints are polled periodically by the host, and have a guaranteed maximum

4 The **International System of Units** (abbreviated **SI** from the French phrase, *Système International d'Unités*)

latency, or time between successive polls, defined in the endpoint descriptor.

A device is not able to perform its normal actions until it is placed in the *configured* state (as will be explained in more depth when appropriate). If the host is unable to guarantee the maximum latency a device endpoint descriptor specifies, it will not place the device in the configured state and hence fail to enumerate it.

2 - Author's assessment of the project

2.1 - What is the technical contribution of this project?

This project implements a USB v1.1 device by means of the Human Interface Device (HID) class which is defined as one of the standard USB classes. Effectively this project developed a means of adding more I/O to a USB-enabled PC without the need for custom drivers (as most operating systems come with USB HID-class drivers) or for user-configuration – this in turn provides the flexibility and portability of USB to almost any potential hardware project; you could plug your device into any host and run your (operating system-specific) application which communicates with your device without needing to fiddle with drivers or settings.

2.2 - Why should this contribution be considered relevant / important to computer science?

This project contributes a USB implementation which makes use of many of the advantages USB offers (communication endpoints, HID report structures, driver-less, bus-powered, no end-user-configuration/plug-and-play), using tools available to hobbyists and educational institutions. As such it offers a modern, end-user-friendly, alternative to legacy ports for hardware projects with great potential to extend the interface to be much more complex.

The micro-controlled hardware interfaced with a Game Boy Advance via UART, and hence demonstrates how existing UART-communications based developments could be (primitively) adapted to USB.

2.3 - How can others make use of the work in this project?

Anyone wishing to create a USB interface to a PC could make use of this work to help them overcome the basic difficulties of working with the PDIUSBD12 chip and USB in general.

Additionally, anyone wishing to create a Game Boy Advance communication link could find enough information in this project to use either “General Purpose Mode” (4-pin bi-directional communications) or “UART mode” of the GBA, as well as the necessary details about cabling and some example code for each mode.

Because standard HID-class reports (data structures defined in a *report descriptor*) are used in this project, data to be transferred has structure, so only this and the `main()` loop need to be adapted to work with a different data format for use in a different application. All the basic functionality to enumerate the device as a HID-class compliant device have been developed, as well as the necessary functions to transmit and receive data as a HID device.

A (.NET-based) Windows XP example application has been used to demonstrate data transfers in both directions, and serves as an example on how to create the PC application that detects and uses a USB device.

2.4 - Why should this project be considered an achievement?

USB is a large and fairly complex standard, with time constraints on certain protocol requests, so implementing it on unfamiliar hardware made debugging tricky and inconsistent at best – verbose debugging would change the timings of signals, producing (and often hiding) errors which would be apparent with less verbose debugging output.

PC-side coding was very limited as the aim of the project was to make a driver-less device, and most coding had to be done within the unfamiliar limitations of a PIC16F877 micro-controller.

Relatively little available documentation on the GBA's communication port made developing and finding source code which made use of it a time consuming process, doing nothing to ease the task of developing an embedded system, PC test-rig and GBA test-rig with so many unknowns at all ends – it was a non-trivial task requiring careful, incremental progress to ensure all three met each other by the end of the project.

2.5 - What are the weaknesses of this project?

Implementing basic USB functionality (such as enumeration) via means of the firmware that was developed took up most of the available time, and as such the demonstration application for the GBA and the PC are simple. Neither demonstration application addressed the original aim of making the GBA a viable PDA.

This project concentrated on the success-path of the applications, and no formal testing of other situations has been undertaken, in-part because I was unable to utilise the USB-IF *Command Verifier* due to compatibility issues I was unable to workaround. While I am confident that my device fails individual requests in a compliant manner I am not confident that the device would pass USB logo certification; however the device has been proven to work on many different Windows XP setups. Further-more no attempts to interface the device with Linux was undertaken.

The firmware cannot handle multiple-transaction reports (data structures), as the RAM requirements of the firmware left less than a full buffer (64 bytes) of free RAM. However, code to handle such reports is in place but cannot be satisfactorily tested, hence this implementation is not suitable for bulk data transfers.

3 - Getting Started

When I originally started on this project I had no special knowledge about USB, little experience in programming my GBA and my most similar undertaking to date was the CG152 project in my first year, where a team-mate and I created "Pong" using a SWET[5] board; nor was I familiar with programming in C – the language I ultimately used to program both the firmware and the GBA.

Because of this uncertainty a very cautious approach of tackling the largest risk first was adopted. The purpose of this section is to show the considerations made during the infancy of the project, and to justify the decisions.

In my original project specification (see section 11.1 on page 62) I identified the following areas for consideration:

1. *Evaluating USB as both suitable for the project, as well as viable/achievable*
2. *Experiment and learn about programming the GBA communications port.*
3. *Choosing a micro-controller on which to base the project architecture*

I shall now discuss each in turn.

3.1 - Evaluating USB

Was it possible to create a USB interface using the tools available to me? First I had to grasp an understanding of USB, which I did by searching the web looking for development sites. I learnt about USB to a level of detail similar to that described in the introduction.

USB certainly offered interesting project opportunities, but was it a feasible project? Searching the web, while revealing much on the USB standard and offering plenty of examples and help with programming the PC-side of a USB implementation, offered relatively little for those wishing to develop the hardware and firmware. However a few resources were available, and among them these remained useful through-out the life of the project:

- USB 1.1 Integrated Circuits and Development Boards
  `http://www.beyondlogic.org/usb/usbhard.htm`
- USB Central
  `http://www.lvr.com/usb.htm`

These websites offered much detail and provided a complete picture of what is involved at every stage between concept and final product.

The first, BeyondLogic.org, detailed a great variety of USB developer boards and integrated

5 SWET – I believe this stands for "Super Warwick Electronic Toy"

circuits designed to take care of the lower-level functions of a physical USB device, such as signalling / transmitting data and interacting with the bus, which was most encouraging. At first I was suspicious of this website which offered so much information (including multi-part article entitled “USB in a Nutshell”) for nothing in return, while offering no product. A bit of research showed that the site is run by a Senior Technician by the name of Craig Peacock, of Flinders University, Australia which set my suspicions at ease. This resource showed me how I might achieve my project goals.

The second, `www.lvr.com`, is run by Lakeview research who publish technical books such as “USB Complete” which I eventually bought and used as my main point of reference through-out the project. The site itself contains (links to) many useful utilities to aid development, such as “sniffers” which allow logging of USB traffic on a Windows host PC as well as example source code. The site and book made it possible to bring all the information together and build the system up using the schematic and source-code from BeyondLogic.org.

A key document at this time was BeyondLogic.org's “USB in a Nutshell – Chapter 7 – PDIUSBD11 and PIC16F87x Example”[3], which detailed an example implementation using a Philips USB IC and a Microchip PIC micro-controller; complete with schematic and source code in C – all other examples I found coded their firmwares in various assembly languages, which I did not wish to spend time learning. Of all the information I found, this example implementation was the most useful and the most relevant.

The Computer Science department have, and use, Microchip PIC micro-controllers frequently, making this document only more relevant – after some research into the Philips PDIUSBD11 was undertaken it was deemed suitable, but unfortunately had also been discontinued. Philips offered the PDIUSBD12 a an alternative, apparently identical in functionality, but using a parallel data bus instead of an I^2C one. The PDIUSBD12 was chosen after careful consideration, but due to the industrial nature of such ICs was only available to buy in batch. The department ordered in the smallest batch we could find: 27 PDIUSBD12 chips.

For details of the PDIUSBD12, please see section *4.3 PDIUSBD12 USB Interface Device with Parallel Bus* on page 25

3.2 - The GBA Communications Port

Through-out the course of this project I did not have access to official developer documentation on the GBA system, as such documentation comes with a development system package which can only be obtained from Nintendo at considerable cost. Instead, I made use of the documentation

provided by the homebrew communities of the web.

My main source of technical information was GBATEK[4], a reputable HTML document within the GBA homebrew community which contains detailed technical information on GBA hardware registers and their usage; however making useful code from this reference often requires knowledge of conventions I was not familiar with.

For example, the entry for the communication-port's general purpose mode is:

> *In this mode, the SIO is 'misused' as a 4bit bi-directional parallel port, each of the SI,SO,SC,SD pins may be directly controlled, each can be separately declared as input (with internal pull-up) or as output signal.*
>
> ***134h - RCNT (R) - SIO Mode, usage in GENERAL-PURPOSE Mode (R/W)***
> *Interrupts can be requested when SI changes from HIGH to LOW, as General Purpose mode does not require a serial shift clock, this interrupt may be produced even when the GBA is in Stop (low power standby) state.*

```
Bit   Expl.
0     SC Data Bit         (0=Low, 1=High)
1     SD Data Bit         (0=Low, 1=High)
2     SI Data Bit         (0=Low, 1=High)
3     SO Data Bit         (0=Low, 1=High)
4     SC Direction        (0=Input, 1=Output)
5     SD Direction        (0=Input, 1=Output)
6     SI Direction        (0=Input, 1=Output, but see below)
7     SO Direction        (0=Input, 1=Output)
8     Interrupt Request   (0=Disable, 1=Enable)
9-13  Not used
14    Must be "0" for General-Purpose Mode
15    Must be "1" for General-Purpose or JOYBUS Mode
```

> *SI should be always used as Input to avoid problems with other hardware which does not expect data to be output there.*
>
> ***128h - SIOCNT - SIO Control, not used in GENERAL-PURPOSE Mode*** *This register is not used in general purpose mode.*

In this entry “134h” is the memory address *offset* from 0x4000000 of the RCNT register, and SC, SD, SI and SO refer to individual pins on the communication port and denote Serial-Clock, Serial-Data, Serial-In and Serial-Out respectively. Not all of this was explicitly in the documentation, and was inferred from looking at the source code of other members of the homebrew community.

Hopeful that I would be able to create a small test application, allowing me to manually set pins logic-high/low or input and display the register contents accordingly, I needed to install an appropriate tool-chain with which to develop the application.

Choosing a tool-chain

Because GBA homebrew has been an interest of mine for some years, I knew immediately that this choice would be between the two most popular development kits: “DevKit Advance” and

"HAM" – either of these would provide me with useful reference source code from others in the community, as well as good community support.

DevKit Advance

A SourceForge[6] project since November 2002, DevKit Advance was the first GBA-development tool-chain I encountered some 3 years back. DevKit Advance describes itself as follows:

DevKit Advance is distribution of GCC, popular among independent Game Boy Advance developers, with the goal of making the creation GBA software as painless as possible for beginners while not getting in the way of the advanced developer.

The main advantage of this tool-chain is its "purity" - it offers a means for compiling C/C++ to a binary files which is in the correct format to be run on actual hardware, all that is required is to include a header files called `gba.h`. Of course this is also its main disadvantage as it does nothing to facilitate the quick development of code.

I originally learnt to develop for the GBA system from a tutorial[7] which used this kit, and so expected that this would be my tool-chain of choice.

HAM

HAM is a newer development kit, with an impressive feature list including a comprehensive library of commonly-used functions for controlling the hardware and even an integrated Development environment!

Illustration 3.2.1 - The HAM Banner

The kit is renowned for being "beginner friendly" as installers are available which require only a few clicks to get everything setup, the IDE supports code completion specific to the HAM library (referred to as HAMlib) and compiling is fully automated through makefiles which are handled by the IDE – additionally it comes with an impressive collection of demonstration applications which show how each feature of HAMlib (and hence the GBA hardware itself) work.

HAM also boasts many convenience features, for instance the IDE has build targets for "Flash, Multi-Boot v2 and Visual Boy Advance" - which are the three main ways of testing

6 SourceForge.net is an on-line collaborative software development management system, often used for open-source projects

7 See `http://www.thepernproject.com`

code: writing it to a flash cartridge (as mentioned in section 1.1), sending it over as a bootstrap via appropriate cable, and running it in a popular GBA emulator for the PC.

The disadvantage of HAM is that it is not open-source; two versions are available: free and registered. The two versions are identical except for a banner (see illustration 3.2.1) – and the associated overhead it requires – which is displayed on all code compiled with the free version. Registering is entirely optional, and costs a nominal €25.

The library contained some very useful functions such as `ham_DrawText()` which works in a similar manner to `printf()` on a PC – the GBA does not have a character-based display meaning text needs to be rendered, making this is a very welcome feature – especially for this project as I intend to do simple code demonstrations and debugging output, so text was all I expected to need.

I was leaning towards HAM at this point because it offered a way to quickly develop. Any source code I found written using HAM was sure to work in my installation, while the same could not be said of DevKit Advance, as my experience with it had taught me, which tends to be customised in hard to duplicate ways by its users. DevKit advance isn't very windows friendly either, and expects bash-type shell.

I chose to go with the HAM system, and have not regretted it.

General Purpose Mode

I quickly developed a simple application which allowed me to set the individual pins to logic-high/low, or to set them as input, and display their logic-levels on the screen at all times and used it to determine the logic-level voltages of the GBA and map SI, SC, SD and SO bits of the register to actual hardware pins.

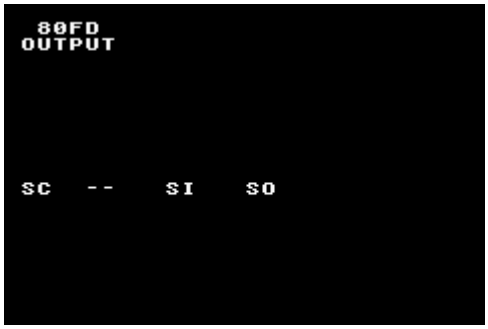

Illustration 3.2.2 - GBA CommProbe

The application was called `CommProbe`, source files and a binary can be found on the attached CD, along with usage instructions (see section 8.1 on page 59).

Pin-out and Voltages

I bought a cheap GBA “link-cable” from a high-street shop and cut it at one end so I could gain access to the wires within. Link-cables are intended for playing multi-player games among 2-4 GBA systems using one of Nintendo's proprietary protocols. Using my `CommProbe` application and a

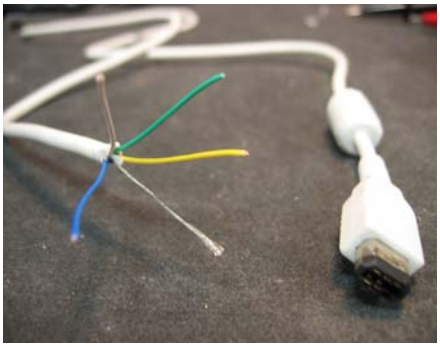

Illustration 3.2.3 - A GBA Link-cable

multi-meter I was able to establish what voltages the GBA considers be to logic high, and map (by colour) the wires of the link-cable to the pins they connect to.

Using data from GBAtek[4] I completed the table with pin names and the role played in UART mode, which I hoped to experiment with later.

Pin# 1 is intended for powering peripherals, and as such there is no connector for this pin on either end of the link-cable, as it is not meaningful to do so. I took readings using a standard GBA and a GBA:SP and noted that the voltages are consistent to +/- 0.05V on both.

When the pins are configured as input, and not connected to anything they float high.

As an aside, I have read that when playing games intended for older Game Boy models (such as Game Boy or Game Boy Colour) the communication port operates at +5V, instead of +3.31V, which is interesting as all Game Boy systems have run on 2xAA (or 2xAAA) batteries.

My obtained results were as follows:

Illustration 3.2.4 - GBA Communication plug and socket

Colour	Pin#	Pin Name	UART Role	Voltage
	1	V_{CC}		
Red	2	**SO** Serial-Out	T_X	3.31V
Orange	3	**SI** Serial-In	R_X	3.31V
Brown	4	**SD** Serial-Data	RTS	3.31V
Green	5	**SC** Serial-Clock	CTS	3.31V
Blue	6	GND		0V

I read a warning about pin# 5 to the effect of "On GBA power-up this is an output, so a 1KΩ resistor ought to be inserted between this and any voltage converter logic to prevent two outputs from shorting each other out" on the Developer's Resource GBA website[8], and this did indeed cause strange behaviour later on the in the project: if the GBA was connected to my project hardware

8 Specifically at `http://www.devrs.com/gba/files/gbadevfaqs.php#PCGCable`

when powering on the GBA, it would often fail to boot and the screen would remain blank, and if the GBA was left attached to the project hardware and turned off it would emit a feint buzzing sound on the speaker, and the older GBA model's power indicator would remain lit – all a result of overlooking this warning.

Running code on a GBA

As described briefly in the introduction, equipment is available to move compiled binary data from PC to the GBA hardware and execute it, either directly via a cable that utilises the GBA's multi-boot capability, or indirectly via a flash-memory based cartridge; which may possibly be written using the prior method.

The most common method is the flash-cartridge based solution as the direct-cable method stores the application in the WRAM[9], both limiting the application size to 256kB and forcing it to be volatile: powering the unit off causes the WRAM data to be lost, and so this method always requires a host PC to send the bootstrap.

Illustration 3.2.5 - A "Flash 2 Advance" cable

Illustration 3.2.6 - The GBA bootstrap loader

From my hobby, I already owned a flash-cartridge set of equipment consisting of a “Flash 2 Advance” cable and “Flash Advance Pro” 256Mbit cartridge and planned to use this for the duration of the project, but the limitations of this approach quickly became apparent. This equipment requires the cartridge be inserted into the GBA when a special bootstrap loader is transmitted from the PC, this loader in turn sends data about the cartridge back to the PC and from then on the PC may read/write to the cartridge via the GBA hardware itself. It is time-consuming (and somewhat error-prone) to transmit the bootstrap, configure the PC-side software, rewrite the cartridge and then restart the GBA hardware to run each new iteration.

9 WRAM: Working RAM – used for general program code/data, as opposed to, say, video data which has it's own specialised RAM (VRAM).

It is usual within the home-brew community to use PC-based GBA emulators to create and debug most code, however this was not an option for me as no emulator exists which adequately emulates the communication port, and testing on hardware would be required at all stages even if one were available.

An oversight in the design of the “Flash 2 Advance” software means it is not possible to send user data as the bootstrap. No alternative software that utilises this system supports such a function, probably because the bootstrap code is stored in the cable's firmware[10]. So I purchased a “Multi-Boot v2” cable (referred to as just “MBv2” in the home-brew community).

Illustration 3.2.7 - A "Multi-Boot v2" cable

The MBv2 cable is designed and produced by an active member of the home-brew community, and offers the ability to transmit user-code through the multi-boot protocol, as well as developer utilities such as an RS232 DB9 connector to allow convenient debugging from the GBA via UART (by providing level-conversion and connector).

By testing development iterations through multi-boot the process is streamlined considerably as only two steps are required to run binary data: reset the GBA and transmit the data to it, the hassle of *resetting, loading, configuring the PC side, writing, resetting, testing, repeating* is avoided. Even the resetting step can be automated if you are willing to solder an extension from the GBA reset line to a pin provided on the back of the MBv2 cable, next to the DB9 connector (see Illustration 3.2.7 - A "Multi-Boot v2" cable, above). The MBv2 also provides some source code libraries for utilising the DB9 connector on the cable from both the GBA and from the PC, even providing a program to interact with the GBA UART mode through the MBv2, allowing GBA UART to be used on a PC with only a parallel port, handy for developing on my laptop!

To summarise, the MBv2 is a developer's cable that streamlines development, while the “Flash 2 Advance” is a tool for storing data on a GBA cartridge. Both are supported by the HAM IDE, which offers keyboard shortcuts to compile and transmit data using either methods.

3.3 - Choosing a micro-controller

Having established that this USB project was feasible, and that GBA communications would be possible at least at a basic level, the last pre-requisite was a micro-controller and the associated development tools, such as chip-programmer and compiler.

10 These cables contain micro-controllers to achieve the precise timings required by GBA protocols.

As it happens, the department's technicians have and frequently use Microchip PICmicro micro-controllers, and have all the necessary equipment, tools and documentation to develop for them using a C-like language. Perhaps more importantly, they also have considerable experience developing for them.

The PIC family used in the BeyondLogic.org schematic, a “PIC16F87x”, is also used by the department in many of the more-involved projects. So a chip suitable for use in the schematic was confidently available while offering a familiar way of coding for it.

I spoke with Mr Rod Moore on the subject of PICs as they relate to my project, and showed him the PDIUSBD12's specification and BeyondLogic's schematic (which used a different USB IC), and he felt confident that the change from I^2C to a parallel bus would not complicate matters considerably. I was confident to use the PIC recommended to me, a PIC16F877, after familiarizing myself with a simpler PIC (a PIC16F84A).

A brief introduction to PIC micro-controllers

A PIC is a PICmicro brand of micro-controller, manufactured by Microchip Technology. Unlike most CPUs, PICs use a Harvard Architecture so data and instruction paths are separate. Different PIC families use different instruction sizes, typically 14 or 16 bits.

PICs use flash memory to store their program, and are traditionally rewritten using devices termed “programmers” in which an inserted PIC's flash memory can be written and read to. Most modern PICs support In Circuit Serial Programming (ICSP) and/or Low-Voltage Programming (LVP), allowing the PIC to be rewritten while permanently wired into its target circuit.

Modern PICs offer a wide range of hardware, such as:

- Timers
- Universal Synchronous/Asynchronous Receiver Transmitter (USART)
- Analogue-to-Digital converters
- Voltage Comparators
- Capture/Compare/PWM modules
- LCD Drivers
- I^2C and SPI peripheral bus support
- Internal EEPROM memory, as a (rewritable) software managed data-store
- Motor Control Kernels

Microchip offer a free IDE, MPLAB IDE (currently v7.10), which lives up to the claim of being easy-to-use. The department have, and use, this development environment with an additional CCS-

compiler add-on which provides a C-like programming language, instead of assembly.

PICs are good value for money[11] and are popular among hobbyists, many of whom share designs and source code of their creations online, encouraging more to join the community.

11 The fairly powerful PIC16F877 used in this project costs around £2.50 at the time of writing

4 - Preparing the Hardware

Knowing that the project is feasible the next step was to prepare the hardware components individually: GBA Communications, Micro-controller & USB IC board and prepare the PC with debugging tools to aid development.

4.1 - GBA UART communications

Experiments in "general purpose" mode had been successful (see section 3.2 on page 14) and I was confident that a solution could be found using this mode – however it would be much simpler if I was able to use the UART mode that the GBA system apparently supports, so I spent some time creating a link between a GBA and a dumb terminal, via RS232.

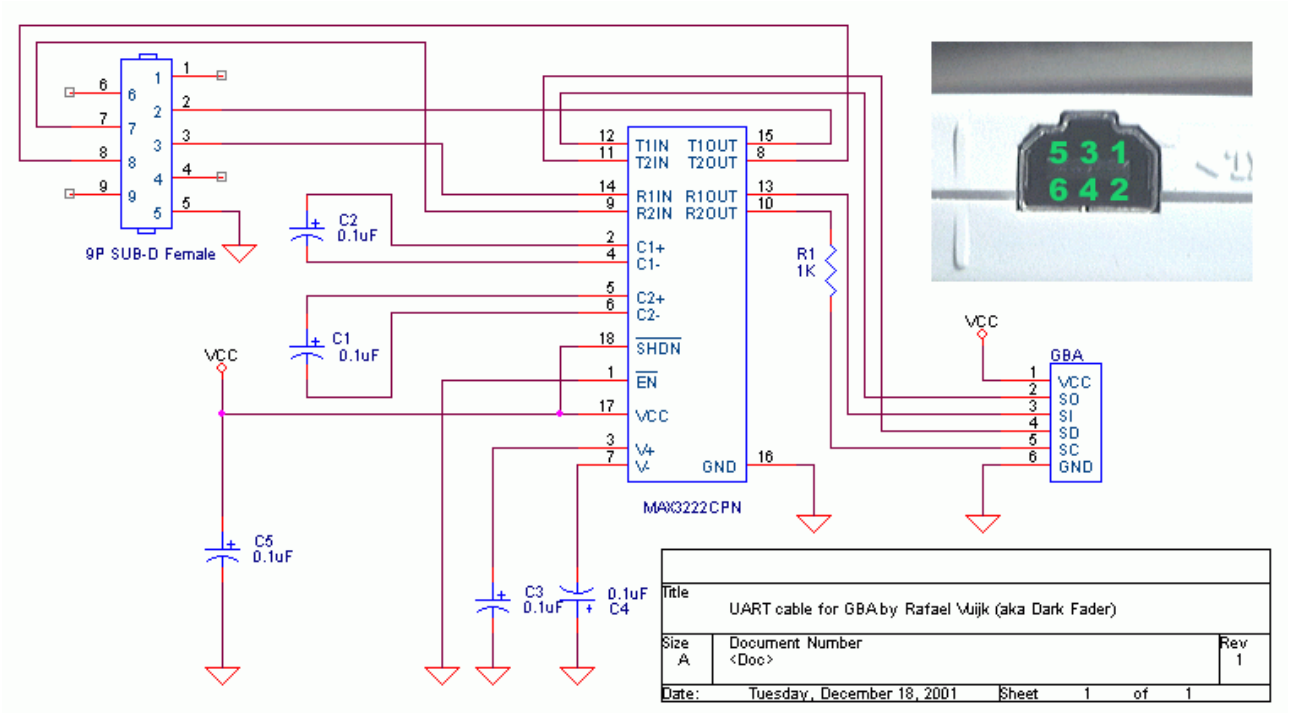

Illustration 4.1.1 - GBA-PC UART Schematic (taken from DarkFader.net)

I found an example implementation of a GBA↔PC connection using UART including schematic, source and binary at DarkFader.net[5], all of which are included on the attached CD (see section 8.2 on page 59). The pin numbering on the GBA used here corresponded with that I had used, as it seems we both based our information on the same source[12].

I realised then that the plug on the link-cable I was using did not connect each pin to a wire in the cable, and the vital `SerialOut` pin was not usable. I acquired another link-cable (an official Nintendo one) and found this to have all pins except power connected, which came as no surprise as

12 `http://www.devrs.com/gba/files/gbadevfaqs.php#LinkPins`

the cable is intended for connecting two self-powered GBAs together.

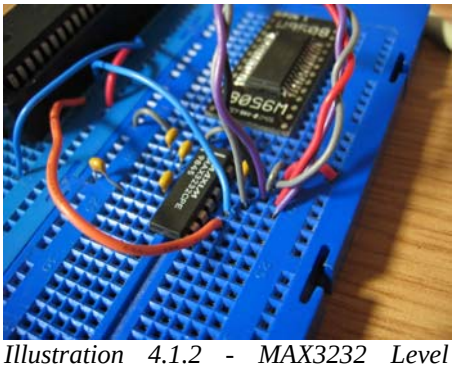

Illustration 4.1.2 - MAX3232 Level Converter

Using a MAX3232 which only has subtle differences from the MAX3222 of the schematic, I built and tested this design which performs level conversion from the near-TTL values the GBA uses (0/3.3V) and the ±12V of RS232, as used on the PC's serial port.

My attempts to use the supplied PC-side application, designed to work with a supplied GBA-side binary, were unsuccessful, as were attempts to recompile the source code. I was able to used the pre-compiled binary on my GBA and use *Advanced Serial Port Monitor* on my PC to send and receive characters to the GBA. Advanced Serial Port monitor performs the same functionality as HyperTerminal, but offers some simple improvements in logging capabilities. However, the GBA binary I was using was designed for code transfer and execution, and not for use as a dumb terminal, so while I was able to verify the connection worked to a limited degree it had no further use. I noted that this configuration *required* the use of the CTS/RTS[13] lines, which would prove inconvenient as the UART libraries of the PIC development tools available to me did not implement these signals.

Thankfully, I discovered another GBA UART demo on FiveMouse.com[6]. This demo mimicked the functionality of a dumb terminal, echoing ASCII characters it received and sending key-presses back. I was able to rework the source code to compile in HAM and quickly made minor modifications to remove the need for CTS/RTS signals.

My `UARTTest` application is included on the attached CD (see section 8.1 on page 59). This application does tend to lose data when doing bulk transfers, due to the time it takes to render characters into the GBA screen-buffer – a simpler application which only echo's data back at the sender does not lose any data, as no internal buffering is done.

This application successfully interfaced with a dumb-terminal, PC serial port and PIC micro-controller during tests.

4.2 - PC Debugging Tools

Before development of the USB peripheral could reasonably be done, I needed to prepare my machine so that I could obtain detailed information about attached USB devices and USB traffic.

13 CTS = Clear-To-Send, RTS = Ready-To-Send

Advanced Serial Port Monitor

Illustration 4.2.1 - A Dumb Terminal

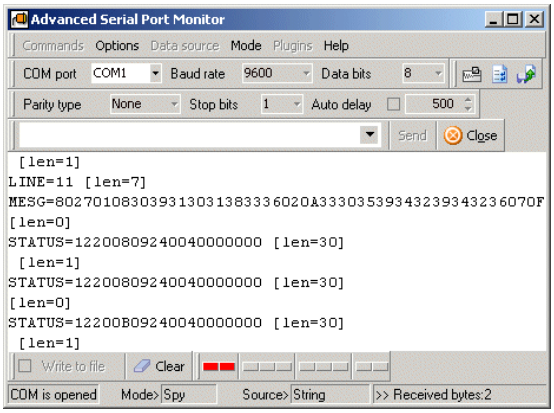

Illustration 4.2.2 - Advanced Serial Port Monitor

As will be seen later, most of my debugging efforts were conducted by a UART connection to the PIC micro-controller, and I quickly found that a dumb-terminal was not adequate as did not offer the ability to scroll back screen-fulls of text. *HyperTerminal* comes bundled with Windows and appeared to offer the ability to scroll back and also to log data to a file, but I encountered problems with it. HyperTerminal would corrupt it's scroll-back buffer rendering it's scrolling ability useless – it would interleave the current screen with the scroll-back data and produce what appeared to be reasonable debug logs, I was only made aware of this problem when using both HyperTerminal and a dumb-terminal at the same time. Additionally, in a frustrating episode, I came to learn the hard way that HyperTerminal ignores all communications in both directions if Scroll-Lock is enabled, without warning.

Advanced Serial Port Monitor[14] offered accurate results, as well as a few niceties such as not locking the file it is logging to, allowing me to manually insert comments and breaks between tests, as well as tagging certain data.

14 `http://www.kmint21.com/serial-port-monitor/`

USB Command Verifier

USB Command Verifier[15] is a tool provided by the USB Implementer's Forum (USB-IF), and provides a means to do preliminary tests of a USB peripheral for compliance with the USB specification, as well as offer a convenient way to thoroughly check the stability of a device. Unfortunately I was never able to get this tool to recognise any USB peripheral at all, including keyboards, mine, web-camera, scanner and USB flash-drive, all of which were verified to be functional.

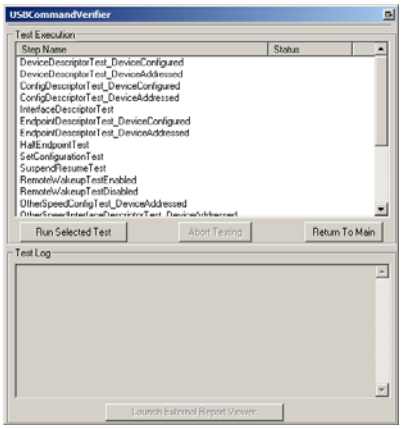

Illustration 4.2.3 - USB Command Verifier

Snoopy Pro

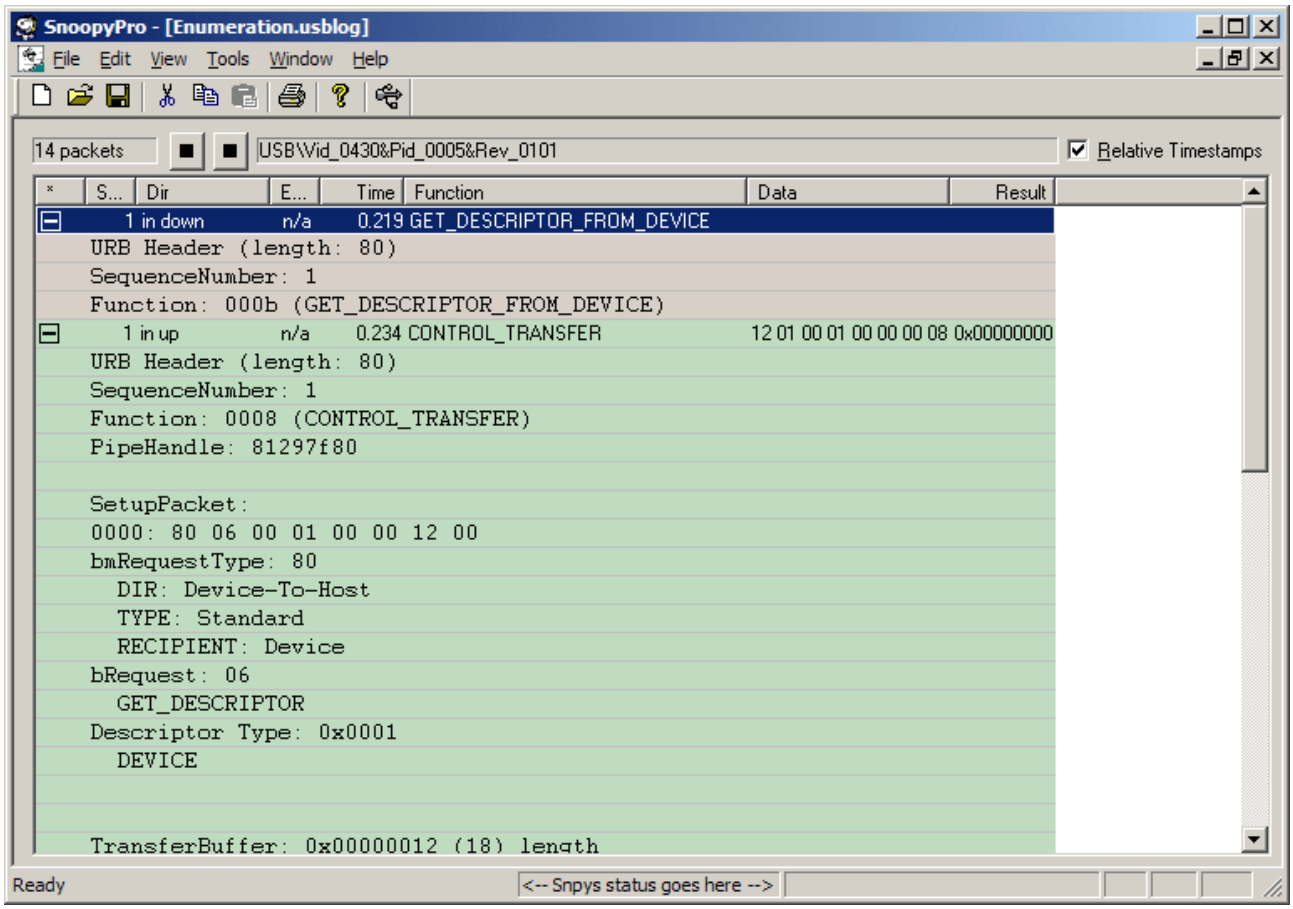

Illustration 4.2.4 - SnoopyPro 0.20 showing an enumeration log

Snoopy Pro[16] is an open-source project to develop a utility for windows which allows logging and analysing of USB traffic between the hardware and device driver. It works well and offers a

15 http://www.usb.org/developers/tools/
16 http://sourceforge.net/projects/usbsnoop/

high level of detail.

Snoopy Pro is unable to capture traffic during the early stages of enumeration and is therefore unable to display useful data logs before the device has been connected to a device driver; until the device is assigned an address during enumeration, there is no log.

A copy of Snoopy Pro 0.20 and some logs have been included on the attached CD (see section 8 on page 59).

HHD USB Monitor

HHD USB Monitor[17] is a shareware application intended for professionals who need to analyse USB traffic, it offers a rich interface and impressive level of detail along with the ability to play back a logged session in (scaled) real-time to view the traffic and get a feel for the timings.

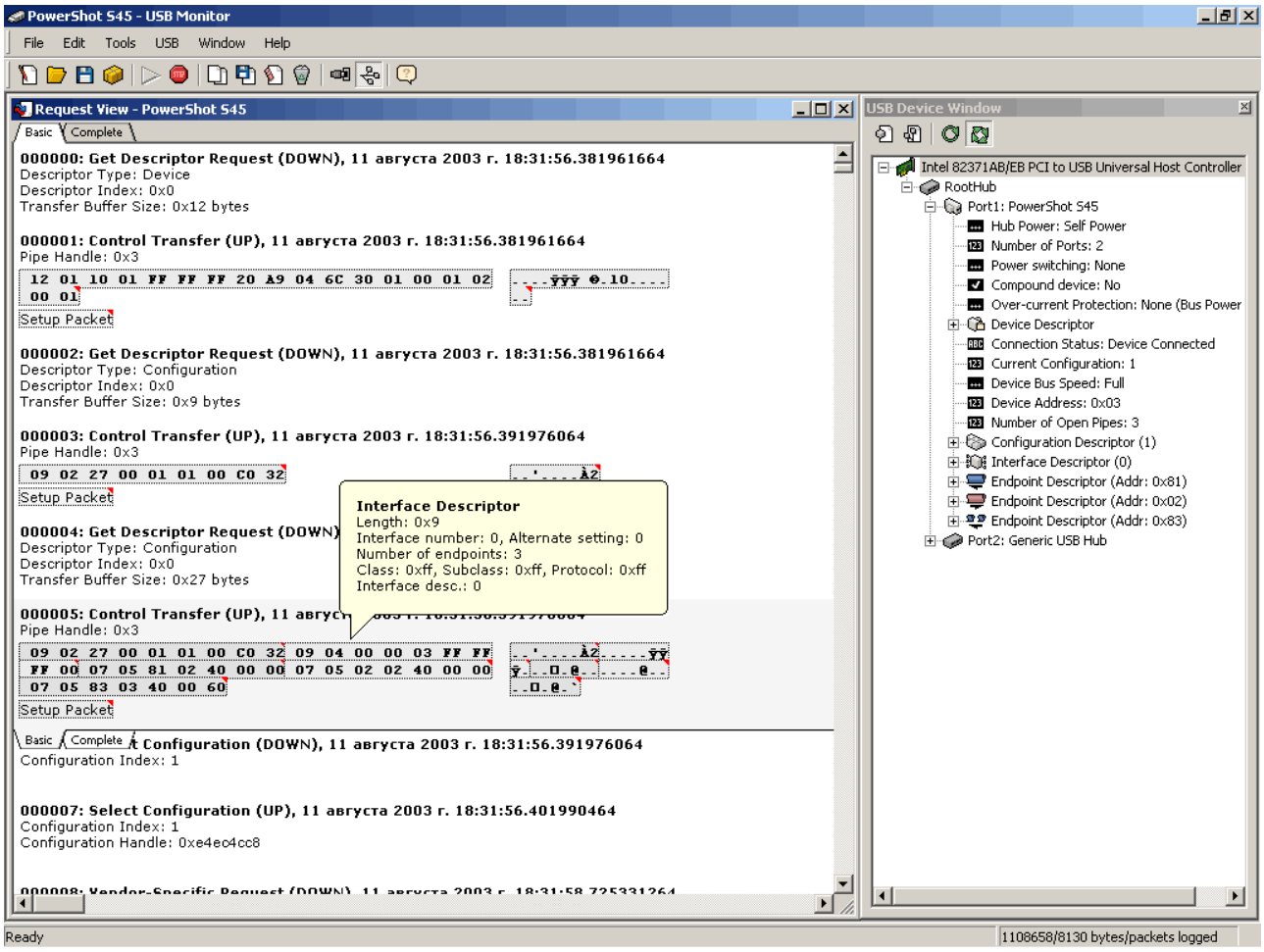

Illustration 4.2.5 - HHD USB Monitor showing an enumeration log

Interestingly, I found Snoopy Pro's simplicity to be an advantage – when I wanted to reverse-engineer some existing USB hardware to get an idea about USB descriptors, HHD USB Monitor

17 `http://www.hhdsoftware.com/usbmon.html`

presented the descriptors as interpreted information, while Snoopy Pro presented them as raw hex with a simple outline above, which I preferred as it facilitated easy duplicating in my source code.

4.3 - System Hardware

In this section I discuss the hardware used, outlining their capabilities and responsibilities as appropriate and detailing the final design which comprised the hardware of this project.

PDIUSBD12 USB Interface Device with Parallel Bus

The PDIUSBD12 is full-speed[18] USB interface device manufactured by Philips Semiconductors and conforms to the *USB specification Rev 2.0 (basic speed)*, it is designed for use in a micro-controlled system as a black-box between the system and the USB bus. As such, it is responsible for interacting with the bus: listening for traffic addressed to it, responding to requests on behalf of the system (via endpoint buffers) and alerting the system (via an interrupt line) when it has successfully received data, or finished sending data.

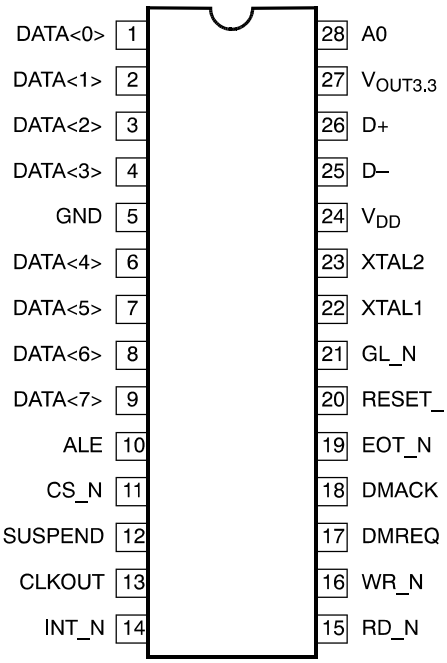

Illustration 4.3.1 - PDISUBD12 pin configuration

The PDIUSBD12 has a 2Mbytes/s parallel bus to connect to a micro-controller, and a maximum achievable throughput of 1Mbyte/s for USB traffic. It supports the four transfer modes of USB: control, interrupt, isochronous and bulk and also provides 64kB double-buffering on it's main endpoint.

Two feature specific to this chip made it stand out among the rest and helped make the decision to utilise it in this project: SoftConnect™ and GoodLink™.

SoftConnect

SoftConnect is a feature whereby the chip's presence of the USB bus can be controlled by software. A full-speed device's presence on a USB bus is indicated by a pull-up resistor on the upstream data line, SoftConnect allows this pull-up resistor to be connected and disconnected from the bus using firmware – useful for bus-powered systems as it allows the firmware to ensure everything is ready for USB traffic to commence before advertising it's presence on the USB bus. It is also part of the specification that an un-enumerated device may consume up to 100mA of power from the bus, and up to 500mA on request, if the host permits.

18 Full-speed as in the USB specification's definitions of low-speed, full-speed and high-speed devices.

GoodLink

A GoodLink pin drives an LED which indicates (without firmware intervention) the status of the connection: If not lit the device is not configured or active, if blinking there is traffic, and if steady it is configured and awaiting traffic.

These features, along with the data sheet[7] led me to believe that Philips had taken care over their design, and kept the developer in mind while doing so – I felt this would be a polished product that I would be happy to work with.

The micro-controller interacts with the D12 in one of three ways: read, write or command. Commands are sent to the D12 by placing the command byte on the bus and then strobing the `A0` pin, the status of a command can usually be obtained by immediately following it with a read of the bus (strobing the `RD_N` pin). When not writing commands, the last selected endpoint's buffer is filled on a write operation.

A good way to get a feel for the responsibilities of the D12 in a device are to view a sample of it's supported commands. The most important are:

- Set Address/Enable: Set the address that the D12 responds to on the USB bus.
- Set mode: Choose which transfer-type configuration of the D12's endpoints are active.
- Read Interrupt Register: Indicates which endpoint triggered the last interrupt (for use with a Read Endpoint Status command), or signals a bus reset / suspend change.
- Select Endpoint: Activate an endpoint buffer for reading/writing
- Set Endpoint Status: Used to (un-)STALL[19] an endpoint
- Acknowledge Setup: When a setup token is received, the D12 flushes its buffers, disables the validate and clear buffer commands and will only re-enable them when this command is performed. This ensures that the firmware knows about the setup token before sending any data.
- Other commands that need no explanation: Read Last Transaction Status, Read Buffer, Write Buffer

A single PDIUSBD12 from RS Components[20] costs £3.63 at the time of writing.

PIC16F877 40-Pin 8-bit CMOS FLASH Micro-controller

The PIC16F877 is a micro-controller manufactured by Microchip Technologies Inc., it is popular among hobbyists and quite adaptable: featuring programmable FLASH memory for user-code

19 STALL tokens from device to host indicate the device is unable to comply and/or does not support the request.
20 `http://www.rswww.com`

(firmware), hardware-USART[21] and I²C for convenient interfacing with other devices, 10-bit multi-channel Analogue-to-Digital converters and 3 hardware timers among other features.

Microchip Technologies Inc. provide a comprehensive data sheet[8] and IDE[22] to aid developers, there is also a wealth of sample source code and schematics available on the web from both Microchip and enthusiasts. The IDE provided allows a syntax very close to the C programming language to be used, something I was most grateful for.

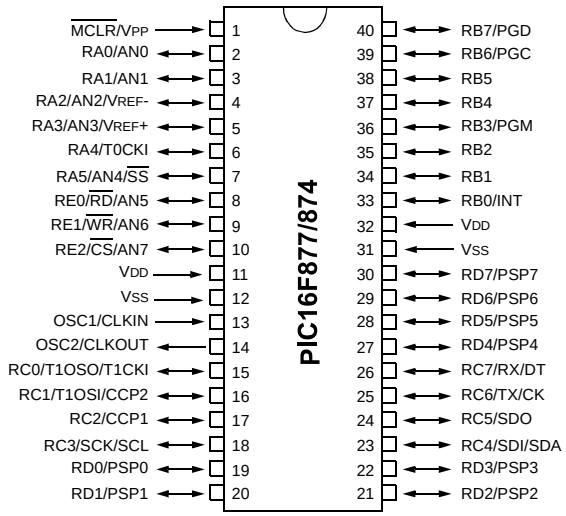

Illustration 4.3.2 - PIC16F877 pin configuration

21 USART: Universal Synchronous/Asynchronous Receiver-Transmitter
22 IDE: Integrated Development Environment

Assembling the Hardware

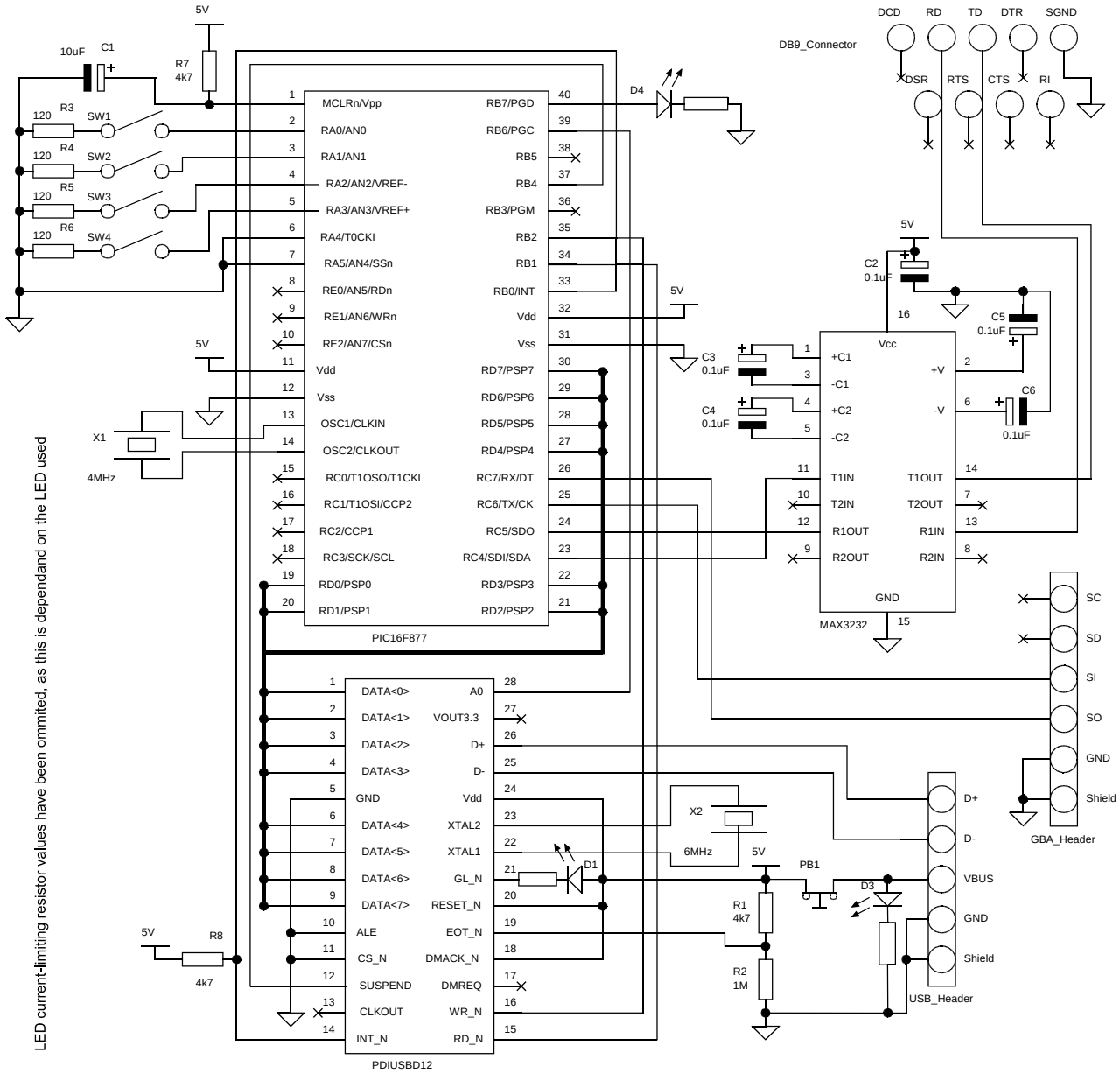

Illustration 4.3.3 - Master schematic for project hardware

Notes:

- This schematic is included on the attached CD in PDF and original *ProSchematic* format, along with a shareware version of *ProSchematic* for Windows.
- A parts list is available in section *11.2 Parts list for Master Schematic* on page 64.
- The data bus between the D12 and P877 connects `Data <0>` to `RD0/PSP0`, and `Data <1>` to `RD1/PSP1` etc.
- LED current-limiting resistor values have been omitted, as this depends on the LED used.

- PB1 (near `USB_Header`) is a break-on-push button intended for resetting the system (see *6.3 PDIUSBD12 Reset issue* on page 48).
- This design can be bus-powered (as it was in this project) or otherwise – the usefulness of PB1 assumes that the system is bus powered – in either case the D12's `EOT_N` must *always* be connected to USB V_{BUS} via the potential divider shown.

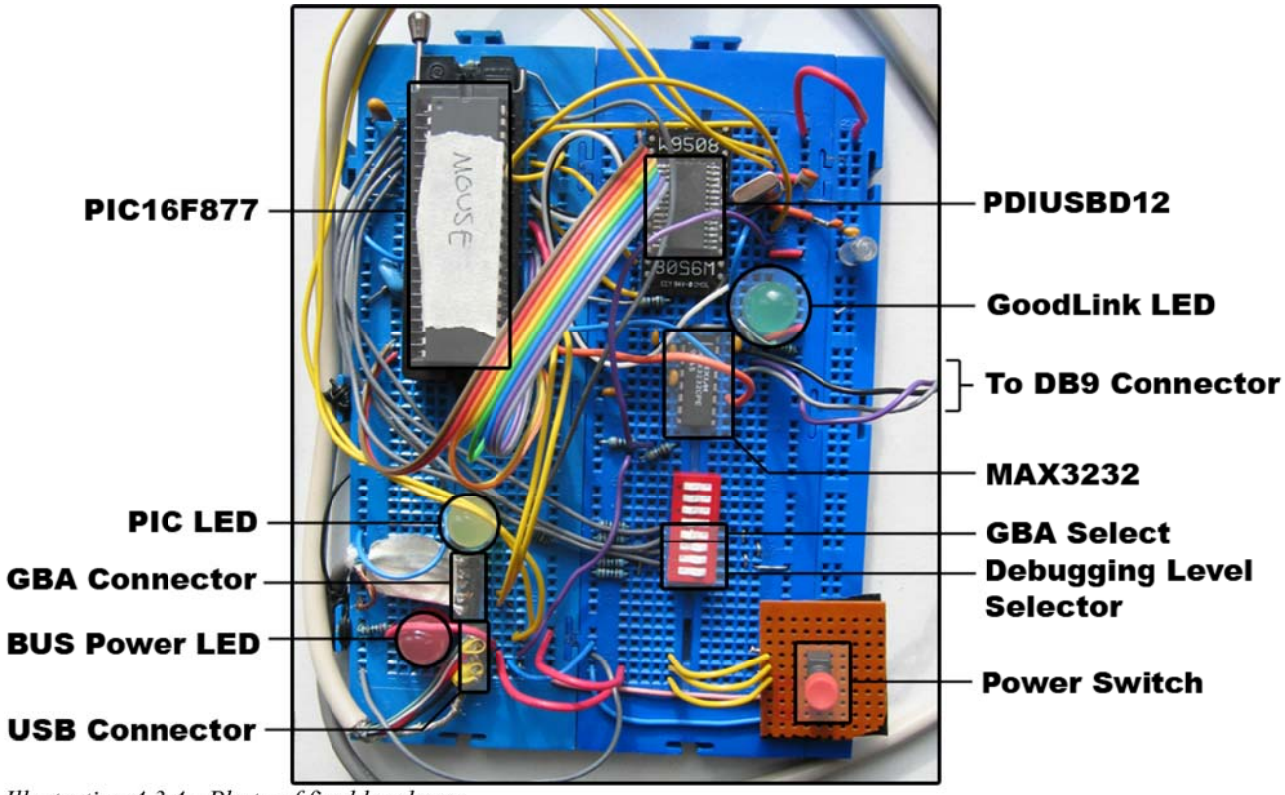

Illustration 4.3.4 - Photo of final hardware

The PIC16F877 and PDIUSBD12 are connected via a byte-wide parallel bus and control lines `WR_N`, `RD_N` and status lines `INT_N`, `SUSPEND`. The D12 is connected to a USB cable with a USB-A male plug on the end (the end required for inserting into a host PC) and hence is connected to a USB bus when the plug is inserted.

The PIC16F877's USART port (pins 25 & 26) is connected directly to a GBA link cable, allowing the P877's USART register to buffer I/O and trigger interrupts on complete reception of a character from the GBA – introducing a level of multi-tasking into the firmware. No 'glue-logic' was necessary as the GBA and P877 TTL[23] voltages were within tolerance of one another.

Another UART connection, implemented in software routines this time, was connected via a MAX3232 RS232 voltage-level converter, creating an interface between the system and a dumb-terminal or PC serial port; and was used for debugging.

23 TTL: Transistor-Transistor Logic

In my implementation the system was bus-powered, so if the system was on (powered) it was safe to assume it was connected to an active host. The only *required* connection to the USB V_{BUS} line is the D12's `EOT_N` pin, which is used for detecting the presence of the bus, aside from this the use of V_{BUS} is entirely optional.

As mentioned in the introduction, debugging was made difficult by the time-constraints imposed by the USB host. Transmitting characters over UART would cause unacceptable delays that often changed the outcome of the device's interaction with the host. Apart from making debugging data as concise as possible (and unfortunately somewhat cryptic) I added a row of DIP switches which would control the debugging level. Three switches were used to enter a binary number in the range 0-7, with 0 being most verbose and 7 being silent except for “uncaught exception” type errors (such as passing invalid constants to a function, indicating a mistake in the program code itself). When the system was reset the DIP switch values would be read in and the debugging level set internally, the DIP switches would not be read again until the next reset.

Another of the DIP switches was used to select if the GBA UART connection should be used for generating report data used later in the project (please see *5.5 Report structures and descriptors* on page 42) or if the data should be randomised – this was a convenience feature used during the final phase of firmware development.

LEDs were used to indicate USB power, PIC status (toggled upon receiving a character from the GBA) and the PDIUSBD12's GoodLink indicator (described on page 26.)

Verifying the System

Using the *Firmware Programming Guide for PDIUSBD12*[9] as a starting point, tests to ensure the the design was valid and usable began, drawing upon firmware source code provided by Philips Semiconductors (see section *8.3 Firmware source code (including examples from Philips)* on page 59).

The sample source code for the PDIUSBD12 appeared to be intended for use with their development board and made assumptions about the attached components (switches, LEDs, etc.) and CPU type; this would have been easy enough to over-come, but the examples were not trivial and the only identifiable implementation was of a Mass Storage-class device. I found the example source code for an HID-mouse on a PDIUSBD**11** to be much more accessible; the differences between the D11 and D12 were almost hidden thanks to good use of hardware-abstraction in the source code.

I noted a few things when testing the system:

- The D12 triggers a bus-reset interrupt as soon as it is powered, the falling-edge of which is usually missed by the PIC which takes longer to start-up.
- When held in reset (noting the issues with resets mentioned previously), the D12's `INT_N` line goes active high, offering a good opportunity to ensure this important line is readable in both states (active low being achieved by bringing the D12 out of reset).
- `RD_N` and `WR_N` must be explicitly opposite logics, it is not sufficient to set one and leave the other inactive.


Illustration 4.3.5 - A Dumb Terminal

At this stage I wrote `GoodLink.c` containing the minimum code needed to make the D12's GoodLink LED light, and was able to successfully read the contents of the interrupt register and confirm the power-on interrupt to be a bus-reset (`0x40`). This code used a UART link between PIC and a dumb-terminal to communicate debugging data and required user intervention to step through each instruction[24]. I was content that the system operated as designed and moved onto writing the firmware.

`GoodLink.c` is included on the attached CD, please see *8.3 Firmware source code (including examples from Philips)* on page 59.

Debugging was done using a dumb-terminal, allowing interaction between myself and the firmware. This proved very useful for stepping through the firmware step-by-step and verifying all logic levels and wire connections on the board, as well as allowing me to view the results of bus-interactions with the D12. This sped up development considerably by allowing a test-first approach to development, implementing each request of the enumeration process as it arose – thus ensuring progress was always been made.

24 Please note that at this time I had the dumb terminal connected to where the GBA-connector is in the schematic.

5 - Learning enough to Enumerate

Content that the hardware was working, it was time to implement the USB protocol requests in firmware. The first step was to get the device to enumerate on the host machine.

5.1 - The USB Enumeration Process

When a device is attached to a host the host learns about the device through a process called enumeration [2]:

> *The process includes assigning an address to the device, reading data structures from the device, assigning and loading a device driver and selecting a configuration from the options presented in the retrieved data.*

During enumeration the device will move through four states defined by the USB specification: Powered, Default, Address and Configured. A typical sequence of events for an enumeration with a Windows host are (adapted from [2]):

1. **The user plugs a device into a USB port.**
   Either directly into the root hub, or anywhere downstream of it. The hub provides power to the port and the device is in the Powered state.
2. **The hub detects the device**
   There is a 15kΩ pull-down resistor on each data-line of the hub's USB port, the presence of a 1.5kΩ pull-up resistor on either line signals a device's presence. if the pull-up is on D+ it is a full-speed device and if D- a low-speed device.
   The hub provides power but does not yet transmit USB traffic as the device will not be ready yet.
3. **The host learns of the new device**
   When the host next polls the hub's status, it will learn of the new device attachment
4. **The hub detects whether a device is low or full speed**
   The hub detects which of the data-lines has the higher voltage when the bus is idle, and hence detect the device speed.
5. **The hub resets the device**
   The host instructs the hub to reset the device. This is done by holding both D+ and D- lines in logic low. These lines usually have opposite logic states.
6. **The host learns if a full-speed device supports high-speed**
   High-speed devices (being the fastest USB 2.0 supports) send special signals during reset which high-speed hubs can detect and respond to. Full and low-speed devices do not send any special signals.

7. **The hub established a signal path between the device and the bus**

   When the hub removes the reset, the device is in the Default state and it's firmware must be ready to respond to control transfers over the default pipe at Endpoint 0. The device assumes the default address of 0x00, the device is now permitted to draw up to 100mA from the bus.

8. **The host sends a Get_Descriptor request to learn the maximum packet size of the default pipe**

   This is sent to the default address 0x00, endpoint 0. The host enumerates only a single device at a time, so only one device will respond even if many were simultaneously attached. The eighth byte of the device descriptor contains the maximum packet size supported by Endpoint 0 (see *5.4 Example: The Device Descriptor* on page 38). Windows hosts request 64 bytes of the descriptor, but as soon as they have received the first 8 bytes (and obtained the maximum packet size) will request the hub to reset the device. The specification doesn't require a reset here, as all devices should be able to abandon control transfers if the host issues another Setup packet, but doing so is a precaution that ensures the device will be in a known state when the reset ends.

   This reset during a control-transfer often leads developers to believe there is a fault in their firmware[2][3].

9. **The host assigns an address**

   Using a Set_Address request. The device reads this address, then acknowledges and stores the new address. The device is now in the Address state. All communications from this point use the new address, and the address remains valid until detached, reset or the host powers down.

10. **The host learns about the device's abilities**

    The host again requests the device descriptor, this time reading all of the response. This describes the maximum packet size for Endpoint 0, the number of configurations the device supports and other basic information about the device, such as supported USB version and vendor ID.

11. **The host assigns and loads a device driver**

    Using this descriptor the host chooses and loads a device driver. In Windows, this is done by trying to match the Vendor and Product IDs and the (optional) Release Number of the descriptor with its driver database. Failing this, Windows attempts to match with any class, subclass and protocol values received from the device. Once loaded is it common for the driver to request the retransmission of descriptors or the transmission of class-specific descriptors.

12. **The host's device driver selects a configuration**

    If the device supports multiple configurations the device driver will select one of these and send

the device a Set_Configuration request, even if the device only supports a single configuration. Once acknowledged by the device, the device is in the Configured state and is now ready for normal use.

Enumeration is complete.

USB STALLs and Bus-resets

When unable to deal with a request (for whatever reason: busy, not understood, etc.) the firmware should stall the Endpoint to whom the request was sent (during enumeration this is always the control pipe, Endpoint 0). This is the compliant way to fail, and was one of the first elements to be implemented.

A USB reset triggers an interrupt from the D12, which reads as a “bus-reset”, similarly the D12 will issue a “suspend-change” when the host tells the device to sleep. This is intended to be used as a power-saving feature and is especially important in bus-powered systems, like the one created during this project, and has strict power-consumption regulations which must be met, however these were never considered in my implementation and the suspend ability of the D12 was disabled by tying its `SUSPEND` pin low. A bus-powered setup was used for convenience in this project.

5.2 - USB Control Transfers

All requests defined in the specification are requested using a control transfer. Control transfers are the most complex type of the 4 transfers modes supported[2].

> *Each control transfer has a defined format consisting of a Setup stage, and optional Data stage, and a Status stage. Each stage consists of one or more transactions that contain a token phase, a data phase, and a handshake phase. Each phase transfers a token, data or handshake packet.*

Control transfers can be used to transfer any data, not just requests defined in the specification. A defined request will begin with the setup stage, alerting the device that this control transfer brings a request defined in the specification. Setup transactions are high priority: if the device is in the middle of another control transfer (including another setup transaction) it must abandon that transfer and respond to the new Setup transaction.

The Setup Stage

The setup stage contains all the information the device requires to complete the request, so the data structure of the data-packet of this stage is of great importance. It consists of eight bytes in five fields: `bmRequestType`, `bRequest`, `wValue`, `wIndex` and `wLength`.

bmRequestType

This is a bit-mapped byte which specifies the direction of data flow for the data stage, the type of request (one of: standard request, class request or vendor-specified request) and the recipient (one of: device, specific interface, endpoint or other element).

bRequest

This is a byte which specifies the request, in the context obtained from `bmRequest`.

wValue

These two bytes (a word) can be used by the host to pass information to the device, such as a new address during a Set Address request.

wIndex

These are another two bytes the host can use to pass information to the device. Typically it is used to pass an index or offset, such as an interface of endpoint number. This data is interpreted in the context of the particular request.

wLength

These bytes specify how much data is to be transferred. When the direction is OUT[25] this is the exact number of bytes to follow, when the direction is IN this is the maximum number of bytes the device may return. The device may return fewer, signalling the end of the transfer by sending a zero-length packet.

The Data Stage

If the control transfer involves a data stage it will consist one or more IN or OUT transactions [2]:

> *The endpoint's descriptor specifies the number of data bytes that each transaction can carry. (For Endpoint 0, the device descriptor specifies this).*

If the `wLength` field in the Setup transaction is 0, there is no data stage: for example the Set Configuration request passes it's data in the `wValue` field of the setup stage and hence does not require a data stage.

Multiple packets are sent as required until all the data has been transmitted, using the largest packet-size the endpoint supports (which is specified in the descriptors, please see section 5.4 on page 37).

25 Recall that all traffic is from the host's perspective: OUT of host, IN to host.

The Status Stage

This is where the device reports the success of failure of the entire transfer. *USB Complete*[2] notes that:

> *In some cases (such as after receiving the first packet of a device descriptor during enumeration), the host may begin the status stage before the data stage has complete, and the device must detect this, abandon the data stage, and complete the status stage.*

5.3 - USB Enumeration Requests

There are 11 standard requests defined in the specification, but only 3 are absolutely necessary to complete enumeration with a Windows host: Set Address, Get Descriptor, Set Configuration.

Each is summarised below (again, descriptions adapted from [2]).

Set Address

Purpose: The host specifies an address to use in future communications

Source of Data: none

Data Length: 0

Contents of Value field: New device address. Allowed values are 1 through 127. Each device on the bus, including the root hub, has a unique address.

Contents of Index field: 0n

Comments: This request is unlike most other requests because the device doesn't carry out the request until it has completed the Status stage of the request by sending a 0-length data packet. The host sends the status stage's token packet to the default address, so the device must detect and respond to this packet before changing its address.

A device must send a handshake packet[26] within 50ms after receiving the request, and it must complete the request within 2ms of completing the status stage. After completion of this request, all communications immediately use the new address.

Get Descriptor

Purpose: The host requests a specific descriptor

Source of Data: device

Data Length: Variable, host requests certain number of bytes of descriptor. If the descriptor is longer that the data length given, return the requested number of bytes only, if equal or shorter send the complete descriptor; in both cases descriptors are sent in one or more packets as needed. If the

26 These are not discussed in this report, they're basic function to acknowledge each transaction phase.

complete descriptor is being sent, and its length is an even multiple of the endpoint's maximum packet size, the transmission will end by transmitting a full packet, but because of this the host will expect more data – therefore the end of transmission must be explicitly signalled by sending a 0-length packet. In a USB transfer additional packets are expected after any packet which is the maximum allowed length (the maximum packet size, defined in a descriptor), hence a packet which is smaller (including empty) will end the transfer.

Contents of Value field: High byte: descriptor type. Low byte: descriptor value.

Contents of Index field: For string descriptors, `LanguageID`, 0 otherwise.

Comments: There are 7 types of descriptor, two of which are only applicable to high-speed devices leaving 5 types of descriptor relevant to this project: device, configuration, interface, endpoint and string. These are covered in *5.4 USB Descriptors*, below.

Configuration descriptor requests are special: the host expects the configuration descriptor, *and all interface and endpoint descriptors it refers to*, to be returned.

Set Configuration

Purpose: Instruct device to use the selected configuration

Source of Data: none

Data Length: 0

Contents of Value field: The lower byte specified a configuration by number. If this value matches a supported configuration the device selects this configuration. A value of 0 indicates *not configured*, if this occurs the device enters the Address state and requires a new Set Configuration request before it is ready for use again.

Contents of Index field: 0

Comments: Upon completion, the device enters the configured state, many of the other standard USB requests require the device to be in this state.

5.4 - USB Descriptors

Descriptors are data structures that enable the host to learn about a device. Each descriptor contains information about either the device as a whole or an element in the device. The higher-level descriptors inform the host of any additional lower-level descriptors. *USB Complete[2]* has this to say :

> *Each device must have a single device descriptor that describes the device as a whole and specifies the number of configurations the device supports and one or more configuration*

descriptors which contain information about the device's use of power and the number interfaces supported by the configuration. Each interface descriptor has zero or more endpoint descriptors that contain the information needed to communication with an endpoint. An interface with no endpoint descriptors can still use the control endpoint for communications.

The standard descriptors are:

Descriptor Type	Required?
Device	Yes
Configuration	Yes
Interface	Yes
Endpoint	No, if the device uses only Endpoint 0
String	No. This is optional descriptive text
interface_power	No. Used if the device supports interface-level power management

String descriptors are used to store descriptive uni-code text to help the *end-user* identify their device on the host, such as the product name and model. These are not used by the operating system in any special way, but are typically displayed by configuration applications, such as Windows' Device Manager and in a pop-up balloon during first enumeration on a Windows XP/2000 system.

Specific classes (including the vendor-defined class) devices may have their own descriptors, which may or may not be mandatory for the class. This provides a structured way to store information needed to ensure ease-of-use and to uphold plug-and-play ideals.

Each descriptor contains a value describing its type, usually followed by the descriptor size (in bytes) and then by a number of fields specific to the descriptor. Of interest to this project are the following:

Type	Value (hex)	Descriptor
Standard	1	Device
	2	Configuration
	3	String
	4	Interface
	5	Endpoint
Class	21	HID
Specific to HID class	22	Report
	23	Physical

Example: The Device Descriptor

The exact format of descriptors is outlined in the USB specification as well as *USB Complete[2]*,

which was the reference I used due its clear break-down and “least-you-need-know” approach. The device descriptor of the project's HID-Mouse implementation is a good example (copied from `HID-Mouse.c`, see *8.3 Firmware source code (including examples from Philips)* on page 59):

```
0x12,       //BYTE bLength
0x01,       //BYTE bDescriptorType
0x10,       //WORD (Lo) bcdUSB version supported
0x01,       //WORD (Hi) bcdUSB version supported
0x00,       //BYTE bDeviceClass
0x00,       //BYTE bDeviceSubClass
0x00,       //BYTE bDeviceProtocol
D12_CTRL_BUFFER_SIZE,   //BYTE bMaxPacketSize (probably 16 bytes)
0x25,       //WORD (Lo) idVendor (Lakeview Research (of USB Complete))
0x09,       //WORD (Hi) idVendor (Lakeview Research (of USB Complete))
0x34,       //WORD (Lo) idProduct (for compatability with HID Class app)
0x12,       //WORD (Hi) idProduct (for compatability with HID Class app)
0x88,       //WORD (Lo) bcdDevice
0x02,       //WORD (Hi) bcdDevice
0x01,       //BYTE iManufacturer
0x02,       //BYTE iProduct
0x03,       //BYTE iSerialNumber
0x01        //BYTE bNumConfigurations
```

Now let's break this down:

```
0x12,       //BYTE bLength
0x01,       //BYTE bDescriptorType
```

The first byte denotes the length of the descriptor (18 bytes), second the descriptor type as defined in the specification (device).

```
0x10,       //WORD (Lo) bcdUSB version supported
0x01,       //WORD (Hi) bcdUSB version supported
0x00,       //BYTE bDeviceClass
0x00,       //BYTE bDeviceSubClass
0x00,       //BYTE bDeviceProtocol
```

The first two of these bytes make up a binary-coded decimal (01.1.0 → USB v1.1), then the device class (not specified[27]), subclass (not specified) and protocol (use depends on class).

```
D12_CTRL_BUFFER_SIZE,   //BYTE bMaxPacketSize (probably 16 bytes)
0x25,       //WORD (Lo) idVendor (Lakeview Research (of USB Complete))
0x09,       //WORD (Hi) idVendor (Lakeview Research (of USB Complete))
0x34,       //WORD (Lo) idProduct (for compatability with HID Class app)
0x12,       //WORD (Hi) idProduct (for compatability with HID Class app)
0x88,       //WORD (Lo) bcdDevice
0x02,       //WORD (Hi) bcdDevice
```

Now the size of the D12's control endpoint buffer (16 bytes, held in a `#define`), followed by the vendor ID, product ID and device code (release version of the product) as another binary-coded decimal.

```
0x01,       //BYTE iManufacturer
0x02,       //BYTE iProduct
0x03,       //BYTE iSerialNumber
0x01        //BYTE bNumConfigurations
```

27 In this example it was specified at the interface level to be a HID-device

And finally, 3 string descriptor indexes (1, 2 and 3 – in this example all three are used, if they were not wanted these bytes should be null) for manufacturer, product and serial number. This is useful, for instance, in a camera to store the camera model: E.G. “MPD-4500 Pro”) and how many configurations the device supports (1 in this example).

The vendor ID must be registered with the USB-IF[28], which costs a $2500 (USD) yearly subscription at the time of writing.

5.5 - The Human Interface Device (HID) Class

There is a lot to the Human Interface Device class, and this section of the report does not pretend to even mention all important details, rather it aims to give a partial understanding sufficient to show the intricacies in developing the system firmware. *USB Complete[2]*:

> *The Human Interface Device (HID) class was one of the first USB classes to be supported under Windows. On PCs running Windows 98 or later, applications can communicate with HIDs using the drivers built into the operating system. For this reason, USB devices that fit into the HID class are some of the easiest to get up and running.*

The name *Human Interface Device* implies that the HID class is for interacting directly with people, and this is indeed the primary intention of the HID class, however it is not limited to this. Any device which can function within the limitations imposed by the HID class is a viable HID device.

The main properties of a HID are thus:

- All transfers conform to a data structure called a report. The structure of a report is defined in a HID class report descriptor, providing considerable flexibility to the developer and potential portability to both the user and developer.
- Each transfer can contain a small to moderate amount of data. For a low-speed device the maximum is 8 bytes per transaction, for a full-speed device (such as the PDIUSBD12) the maximum is 64 bytes per transaction, and for a high-speed device 1024 bytes per transaction..
- The device may send data to the host at unpredictable times: there is no way for the computer to know when the user will press a key on the keyboard, so the host's driver polls the device periodically to obtain new data.
- The maximum speed of transfers is limited. At full speed the host guarantees no more than 1 transaction per millisecond for a maximum transfer rate of 64 kB/s.
- The *rate* of transmission is not guaranteed, only the latency (i.e. polling interval). For instance if a device is configured for 10ms intervals, the time between transactions may be any period

28 USB-IF: Universal Serial Bus Implementers Forum

equal to or less than this.

There is a single exception to this. Devices configured to transfer data during every frame at full speed have the lowest possible latency, and hence are guaranteed to be polled at an exact interval thereby achieving a guaranteed rate of transfer.

HIDs primarily transfer data from device to host, but the opposite is possible – the classic example is a joystick with force-feedback. All HID transfers use either the default control pipe, or an interrupt pipe on another endpoint. HIDs are required to have an interrupt IN endpoint for sending data to the host, and interrupt OUT endpoint is optional. This was not the case in USB v1.0 and so Windows 98 Gold, the first version of Windows to support HIDs, does not support the interrupt endpoint but instead uses the USB v1.0 *Get Report* control request. Use of this request is discouraged in the current version of USB.

An interrupt endpoint is best for low-latency data which is time-critical (such as movement data from a mouse), while control transfers are better for transferring configuration data or similar which is not time-critical.

Because HIDs are defined on a per-interface basis it is possible that a HID be only part of a USB device, for instance a video player may also implement a HID interface for software control of the machine's functionality.

Firmware Requirements

Device descriptors must identify at least one HID class interface and support an interrupt IN endpoint, in addition to the default control pipe. It must also contain a report descriptor which defines the data structure used for transmitted data (termed *reports*).

To send data the device is required to support Get Report requests which use control transfers to transmit data, and interrupt IN transfers which do not use requests at all (interrupt data is buffered internally and transmitted when the host next polls the device). To receive data it must support Set Report control transfers and, optionally, interrupt OUT transfers.

As mentioned previously, all HID data are transmitted in data structures termed “reports”. These reports mus be specified in the HID report descriptor using a format defined in the HID specification. Devices may support one or more reports (i.e. can have multiple data-structures).

Reports come in three flavours: Input, Output and Feature. As with all USB traffic, this is from the perspective of the host – Input reports go from device-to-host, Output from host-to-device and Feature reports can go in either direction.

Report structures and descriptors

At the core of the HID class is the report and its structure. Report structure is defined in the report descriptor, using a flexible format defined by the USB-IF. The format was designed to be compact, at the expense of complexity and loss-of-readability.

A report descriptor is a type of class descriptor. The host retrieves the descriptor by sending a Get Descriptor request with the value field of 0x22 in the high byte (indicating a HID report) and the report ID in the low byte (usually 0x00).

The format used when creating report descriptors defines fields within a single byte to specify a property along with the number of following bytes which describe the value of this property. This format has dynamic-length user-definable properties.

Perhaps it is clearest to show a well commented report descriptor, as used in the HID-Joystick implementation created in this project:

```
0x05, 0x01,                    // USAGE_PAGE (Generic Desktop)
0x09, 0x04,                    // USAGE (Joystick)
0xa1, 0x01,                    // COLLECTION (Application)
0x09, 0x01,                    //   USAGE (Pointer)
0xa1, 0x00,                    //   COLLECTION (Physical)
```

This defines the report to be a Joystick, and allows the host's operating system HID driver to load any appropriate modules to interpret the rest of the structure.

```
0x05, 0x09,                    //     USAGE_PAGE (Button)
0x19, 0x01,                    //     USAGE_MINIMUM (Button 1)
0x29, 0x03,                    //     USAGE_MAXIMUM (Button 3)
0x15, 0x00,                    //     LOGICAL_MINIMUM (0)
0x25, 0x01,                    //     LOGICAL_MAXIMUM (1)
0x95, 0x03,                    //     REPORT_COUNT (3)
0x75, 0x01,                    //     REPORT_SIZE (1)
0x81, 0x02,                    //     INPUT (Data,Var,Abs)
```

This defines 3 1-bit buttons. Specifying logical properties allows scaling – for instance: if the application expects a pressure the button could be scaled to be the maximum possible pressure by setting its logical maximum appropriately.

```
0x95, 0x01,                    //     REPORT_COUNT (1)
0x75, 0x05,                    //     REPORT_SIZE (5)
0x81, 0x03,                    //     INPUT (Cnst,Var,Abs)
```

Here some some generic data is defined. This case it is used as padding to complete a byte (3x 1-bit + 1x 5-bit).

```
0x05, 0x01,                    //     USAGE_PAGE (Generic Desktop)
0x09, 0x30,                    //     USAGE (X)
0x09, 0x31,                    //     USAGE (Y)
0x15, 0x81,                    //     LOGICAL_MINIMUM (-127)
0x25, 0x7f,                    //     LOGICAL_MAXIMUM (127)
0x75, 0x08,                    //     REPORT_SIZE (8)
0x95, 0x02,                    //     REPORT_COUNT (2)
0x81, 0x06,                    //     INPUT (Data,Var,Rel)
```

Here we define the axis of the joystick, and scale it to ±127, using a byte for each axis.

```
0x06, 0x00, 0xff,              //      USAGE_PAGE (Generic Desktop)
0x09, 0x01,                    //      USAGE (Vendor Usage 1)
0x75, 0x08,                    //      REPORT_SIZE (8)
0x95, 0x02,                    //      REPORT_COUNT (2)
0x91, 0x02,                    //      OUTPUT (Data,Var,Abs)
```

This defines a new entity for transmitting two bytes from host-to-device. This is not needed by joysticks, nor does it interfere with them. It was added to demonstrate my implementation of Set Report control transfers in the firmware, coupled with a PC application which sent data. This is an example of a class-extension, it does not contradict or affect the normal functioning of the joystick class, but provides additional functionality to software that supports it. It is interesting to note that this extension does not require special drivers, as report data is available to user-mode[29] application through a Microsoft HID API; instead an application with special support is required.

```
0xc0,                          //    END_COLLECTION
0xc0                           //  END_COLLECTION
```

These bytes close the collections and end the report descriptor.

In reiterate: HIDs use interrupt transfers to transmit structured data as defined above, with a guaranteed latency.

See *8.3 Firmware source code (including examples from Philips)* on page 59 for the complete source.

29 Windows systems make a distinction between system and user components, the latter not being permitted to access hardware or memory directly.

6 - Firmware Design and Development

As per the recommendations of the Philips *PDIUSBD12 Firmware Programming Guide*[9], the firmware was designed around a central Interrupt Service Routine (ISR) which would be triggered by a falling edge on the `INT_N` line of the D12.

6.1 - The USB Interrupt Service Routine

The USB ISR of my firmware functions as a large while-loop until the D12 interrupt is cleared, indicated by a high `INT_N` pin, this is necessary because the P877's external interrupt is triggered by an *edge*, and it is possible that additional interrupts be triggered while one is being serviced.

After verifying that there is an interrupt which requires servicing, the D12's interrupt status registers are read into P877 memory and used to determine the cause – a bus-reset, suspend, or endpoint. If the interrupt is not an endpoint interrupt the action of reading the register clears the interrupt. If it is an endpoint, the endpoint's *last transaction status* register is read to further determine the cause and clear the interrupt. An endpoint interrupt will occur upon successful transmission/reception (depending on the direction of the endpoint), but can also occur upon transmission errors if requested. In normal operation the D12 retransmits upon a failure automatically and does not trigger an endpoint interrupt. Rather, interrupts are used to notify the micro-controller that the send buffer is empty (and ready to be filled) or that the receive buffer is full (and ready to be read and emptied).

For this project I implemented a HID-class device, which requires the use of an additional interrupt endpoint (the D12's Main OUT was used) for interrupt transfers, using the control-endpoint for class-requests.

Sending data was achieved by selecting the appropriate endpoint on the D12 and then writing the data to the D12 (filling an internal double-buffer), and (once done), flagging the buffer as valid and ready for transmission. The D12 took care of responding to polls from the host by using this buffer, signalling the P877 when the buffer was empty (and switching to its second internal buffer in the case of Main OUT[30]). Receiving data was done in a similar fashion, reading the buffer out of the D12 into the P877 RAM and then flagging the D12 buffer as read and empty.

Upon receiving an unimplemented request the firmware would stall the endpoint, thereby failing in a compliant manner.

(A copy of this digram in included in PDF format on the attached CD, as described in *8 About the attached CD* on page 59.)

30 The D12 utilises a transparent double buffering scheme on its main pipe (endpoints 4 & 5)

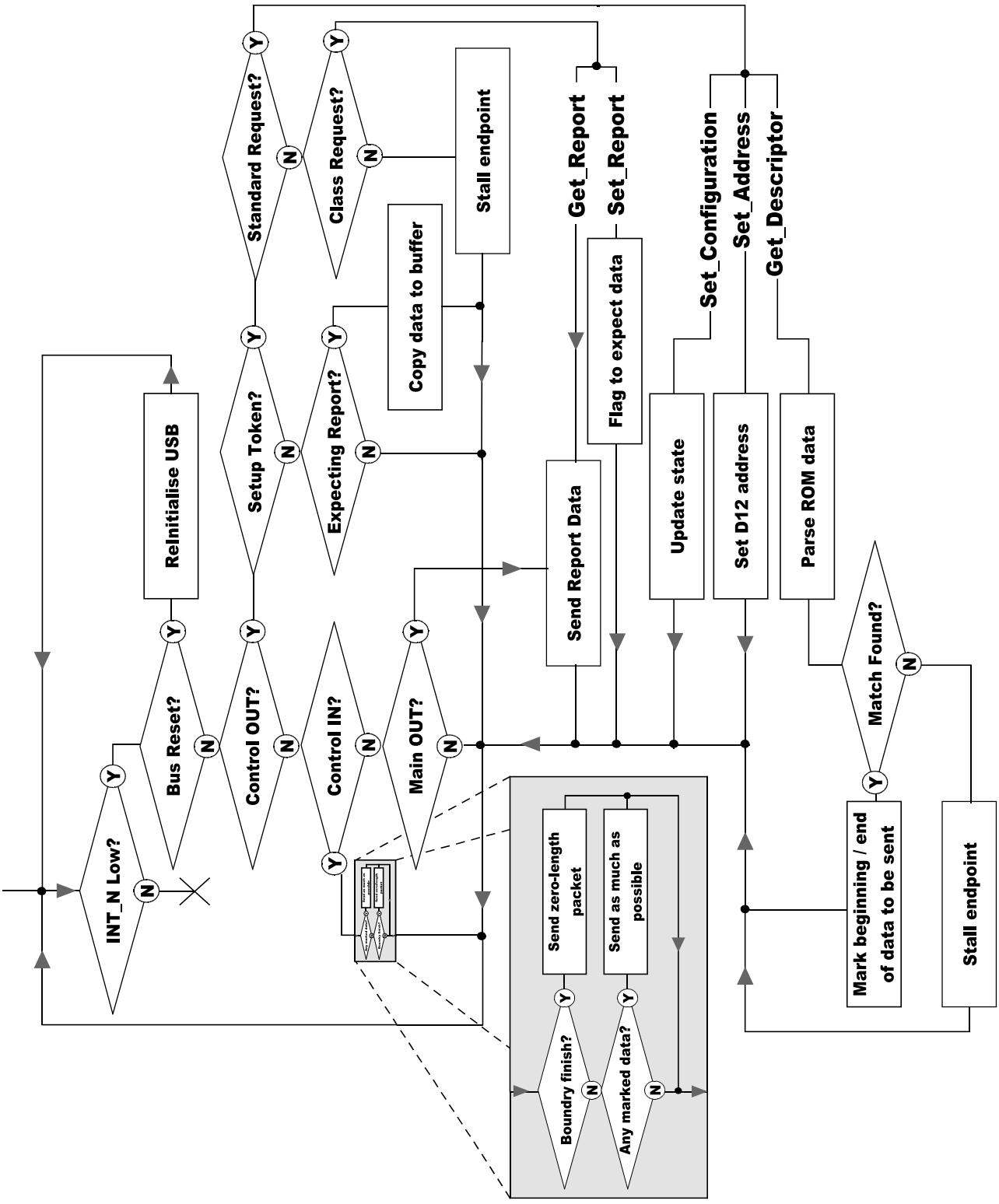

Illustration 6.1.1 - Flowchart of the Firmware's USB Interrupt Service Routine

This diagram is not intended to be complete, but to show the high-level structure of the firmware's interrupt service routine. The entry point is the incoming line at the top left, and the exit point is just below it on the "**N**" branch of "**INT_N Low?**".

6.2 - Development

After some notable difficulties (see *6.3 Notable problems encountered* on page 47) the firmware successfully enumerated a HID device, which would fail to start (the HID components not yet being implemented). See `Enumerated.c` and `Enumerated II.c` (details at end of section)

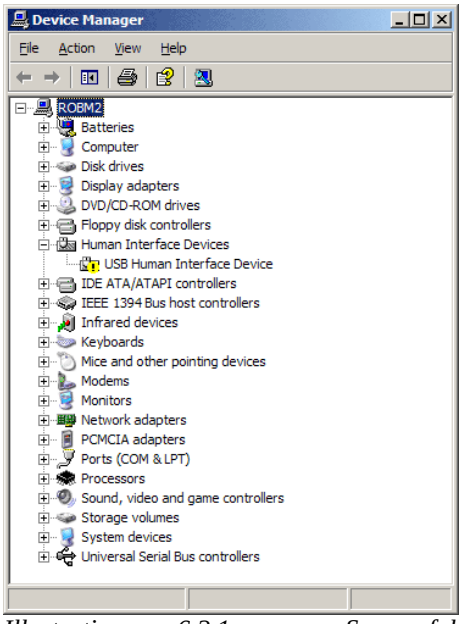

Illustration 6.2.1 - Successful enumeration

This was the first proper feedback to be received from Windows, and from this point development sped up considerably, concentrating on implementing request handlers and adding functionality, instead of fiddling with settings in the dark.

Soon enough a working implementation of a HID-Mouse (with the string descriptor "`MeerMouse`") was completed, transmitting identical report data on each interrupt. The report descriptor defined positional data to use relative distances (instead of absolute coordinated), which caused the cursor to crawl steadily along the screen. From there it was little work to have the mouse moving in all directions, then activating buttons, and from there using the GBA (via USART interrupts) to control this data by means of the `UARTTest` application. See `HID-Mouse.c`

However, it was impossible to test host-to-device report transfers, as mice (and keyboards) are locked by the system on Windows for security reasons, this meant that a user-mode application cannot obtain a handle to the device nor interact directly with it.

HID-Mouse was converted to HID-Joystick, which uses a near-identical report structure but does not suffer a system-lock. A test application obtained from `www.lvr.com/usb` (written on the .NET framework) allowed the firmware's implementation of the Set Report request to be quickly tested, and was demonstrated during the project presentation, the joystick movement being easily observed from the *Game Controllers* panel in *Control Panel*. See `HID-Joystick.c`.

Source code for all check-points and implementations is included on the attached CD, please see *8.3 Firmware source code (including examples from Philips)* on page 59 for details. The PC-side HID-class test application is also on the CD. The default `VendorID` and `ProductID` that the application starts with match that of the device created.

6.3 - Notable problems encountered

During development of the firmware, some unforeseeable problems arose which may be of interest to anyone wishing make use of / further develop this project. Many of these issues have already been mentioned in relatively little detail.

PIC16F877 Memory Limitations

The PIC16F877 micro-controller provides 368 bytes of data memory[8]. This amount of memory proved limiting: the USB descriptors used in this project take up 174 bytes, accounting for almost half the available data memory. Local (P877) endpoint buffers require 16 bytes for each direction of the control endpoint and up to 64 bytes for the larger interrupt OUT buffer used for sending report data, bringing the total used RAM before writing the first line of code to 270 bytes – almost 75% of the available memory!

During early development of the enumeration code the compilation error “Out of Memory error for target device” occurred, at this time the firmware did not allocate a buffer for the interrupt OUT endpoint (the HID class was not yet implemented).

My solution was to store the descriptor data in program memory instead of data memory. This is possible through the `TABLE` construct provided by the CCS-compiler add-on for MPLAB that I used to develop the firmware. A major limitation of this construct is that data may only be accessed using an offset from its beginning, like an array but without the ability to use array pointers. This is because access to program-memory requires certain pre and post operations to be performed in the CPU – operations not required by normal memory access – hence the CCS compiler cannot allow pointers to, or dereferencing of, `TABLE` data.

I restructured my descriptors from being individual arrays to being a large character `TABLE`, taking precautions to ensure they could be separated again. The configuration descriptor does not feature a size byte, but a `wTotalLengh` word instead which indicates the length of the configuration descriptor *and* all sub-descriptors it references. Without a size byte it would be impossible to separate the configuration descriptor from those which come after it, so I prepended it with a size byte which my descriptor separating algorithm would not transmit when responding to a Get Descriptor request for this descriptor.

This static descriptor data would be searched, using a parsing approach, by reading the first 2 bytes of every descriptor. The first byte indicates the type of descriptor and hence checks if this is the descriptor requested by the host. The second byte indicates the descriptor size which is used to discern at what offset the next descriptor begins. After accounting for exceptions (omitting the size

byte of the configuration descriptor, and checking string descriptor indexes[31]) the new solution was both much more memory-efficient and also much more compact, readable and elegant when compared to the array-pointer solution used previously.

Debugging issues

Debugging needed to be done at the device as no useful data is available from the host before enumeration completes. This was achieved by using the PIC16F877's hardware USART to transmit text using `printf()` statements in the firmware.

This introduced sizeable delay, as each character takes considerable time to transmit. At a baud rate of 9.6kbps it takes 0.104ms per bit, including stop and start bits, resulting in a time of 1.146ms to transmit a single character. The fastest reliable baud rate achievable on a PIC running at 4MHz (according to the CCS-compiler I used) was 19.2kbps, resulting in a time of 0.573ms per character.

This delay would alter the outcome of USB bus transactions. If the D12 does not have data to send when the host requests some, it will return a STALL token, after receiving 3 or more STALL tokens the host will give up and the transaction will fail.

The simplest way to remove these delays would be to remove all `printf()` statements and avoid them entirely, but this was not an option. Instead I aimed to avoid delays whenever possible by adding “debugging levels” to the firmware. I connected a set of DIP switches to the PIC, the values of which would be read on start-up, dictating the debugging level. Three switches were used to indicate how detailed debugging should be, offering a range of 0 (most verbose) to 7 (only display errors in the firmware itself). All debugging statements in the firmware would compare their debugging level with that of the current execution, and only display if their level was equal or higher.

This worked well, level 7 debugging, the most concise, would be used until a problem was encountered at which time the debugging level would be lowered (and hence made more verbose) and then the firmware restarted displaying more detail at the expense of a slightly altered execution time. This would be repeated until enough detailed was gathered to resolve the issue.

Some errors could not be debugged using this approach, as the different timings of debugging levels would alter the outcome of the execution.

PDIUSBD12 Reset issue

Unfortunately, it was not possible to reset the D12 using software control on the P877, as the D12 would exhibit strange behaviour after receiving the reset-pulse on its `RESET_N` pin.

31 As shown in *5.4 Example: The Device Descriptor* on page 38, a device may have multiple string descriptors.

Eventually I discovered a note in the PDIUSBD12 FAQ[10] on the Philips Semiconductors website:

> *Q: What should be the width of the reset pulse PDIUSBD12?*
>
> *A: The external reset pulse width has to be 500 µs (min.). When the reset pin is LOW, make sure the CS_N pin is in the inactive state; otherwise, the device may enter Test mode.*

Because `CS_N` is permanently tied low in my setup, the D12 would always enter test mode when I wanted it to reset, and I was unsuccessful in ensuring `CS_N` was in the inactive state during a reset despite efforts. After spending considerable time on the issue, I cut my losses and instead added a push-button which breaks the circuit between V_{BUS} and the reset of the system when pushed. This provided a convenient way to reset the entire system during debugging, which did not wear out the tracks of my USB plug and socket (I was unplugging and reinserting constantly until the addition of this button).

Undocumented PDIUSBD12 Set Address command behaviour

As described in *5.2 USB Control Transfers* on page 34, standard USB requests should be performed by the device immediately, and the status stage (acknowledgement) initiated after it has completed. The exception to this rule (which is heavily emphasised in all the literature I encountered, including *USB Complete[2]*, my main reference) is the Set Address request.

When servicing a Set Address request, the device must buffer the address, initiate and complete the status stage, and *then* change address and wait for another request (as the device will be in the Addressed state, and should not perform any of its designed actions until it is in the configured state).

The PDIUSBD12 implementation of the Set Address does not set the address immediately, as the data sheet[7] implies:

> *This command is used to set the USB assigned address and enable the function.*

I discovered from the USB-IF's Developers [sic] Discussion Board[32] from someone in a similar position to me that:

> *The 'D12 has special support built in to it for changing the address value at the correct time.*
>
> *Issuing the SET ADDRESS ENABLE command (0xD0) to the 'D12 only updates a holding register…*
>
> *The D12 will wait until the appropriate time (right after the zero length packet representing the STATUS phase gets IN'd by the host) before it actually switches from the default address (0) to the new address.*

32 `http://www.usb.org/phpbb/`

This proved to be correct, by amending my firmware to perform the Set Address D12 command *before* initiating the status stage the device became considerably more compatible. Until this change the device would enumerate on my development PC but would fail to enumerate on any other machines I attempted (all Windows XP SP2 systems), once this change was made the device would enumerate on all machine I tried it on.

I believe the presence of two USB sniffers (see *4.2 Snoopy Pro* and *4.2 HHD USB Monitor* on page 23) on my development PC relaxed the timing restrictions of USB by incurring soft delays, thereby allowing the device to enumerate while other PCs were not as accommodating.

7 - Evaluation

7.1 - Did the project achieve its goals?

The goals of this project, as described in the original project specification[11], para-phrase as follows:

> *To create an interface between a PC and a Nintendo Game Boy Advance (GBA) system which is easy to use (specifically: requires no special drivers, and no user configuration) and to create some proof-of-concept software that shows the use of this interface to communicate between the two.*

No special drivers required for the PC

A USB HID device was created which successfully interfaces with a PC. The device requires no system drivers when the host operating system is Microsoft Windows 98 or above, as these systems are shipped with native USB Human Interface Device class support. Windows 98 Gold[33] supports and old version of the USB HID class, which does not permit an interrupt endpoint, now preferred by the current version of the HID class. Instead, Windows 98 Gold uses the control pipe with a Get Report request to obtain report data.

This request is still a mandatory part of the HID specification, and an implementation of it exists in the firmware I developed, but it could not be tested adequately. Although the HID Tester application on the attached CD[34] has an option to "Use control transfers only", tests of this feature showed the application to receive the correct amount of data, but all null. This is believed to be due to a conflicting Set Report request being issued to the device (by the application) simultaneously.

The scope of this project was limited to working with Windows systems, and in that respect it was successful. No special drivers are required for the device to enumerate and be accessed by user-mode applications.

Linux support has not been considered.

No special drivers required for the GBA

As the GBA's hardware-implemented UART support was utilised for this project, it is fair to say that no special drivers are needed to operate the GBA when connected to the device. However, as in the PC's case, software designed to work with the device is required. In this case any software which uses the GBA's UART connection at 9600 baud and disables flow control (CTS/RTS) can communicate with the device. For that matter, any UART device capable of the above can be used in the GBA's place – for instance a dumb terminal.

33 Windows 98 Gold was the first release of Windows 98, "Gold" is used to disambiguate it from "Second Edition"
34 See *8 About the attached CD* on page 59

No configuration required

Device-PC Interface

While the HID tester application offers the ability specify the `VendorID` and `ProductID` of the device, in practice this would be coded into the application and not available to the end-user to change, or would be acquired through a Windows API if looking for any attached device from a class of devices.

Hence there is nothing to configure, the enumeration process negotiates an address for the device, and software acquires a handle on the device via its `VendorID`, `ProductID` and (optionally) its `DeviceID` (release number), or again through an API call if only the device type is important. The user is only required to attach the device and start an application which utilises it.

It is true to say that no configuration is required to use the device when attached to a PC.

GBA-Device Interface

The UART baud rate (9600) utilised between device and GBA is non-negotiable, and both the are coded to use the same settings. However, if the device was to be used as a generic GBA↔PC interface, it is unlikely that these settings will be coded into the GBA, and in this respect this goal was not achieved.

Overcoming this would require creating a negotiation protocol, similar to the process of enumeration, for the GBA. Such a task is non-trivial, beyond the scope of this project, and as such has not been attempted.

It was not the aim of this project to implement a serial port over USB, and so I do not consider this failure to be significant.

Proof-of-Concept Applications

Two applications were needed to effectively demonstrate the working system, one on the GBA and one of the PC.

GBA `UARTTest`

A dumb-terminal like application for the GBA was developed during the course of this project, which effectively demonstrates use of UART communications on the GBA. Received data is displayed as hexadecimal on the screen, and button presses are transmitted as ASCII characters (for instance up is transmitted as '^', down as 'v', left as '<' and so on).

The GBA interfaces with the device via UART and so this application is all that is needed

at the GBA to demonstrate working USB communications.

This application (including source code and binary) is included on the attached CD (see *8.1 Game Boy Advance Applications* on page 59).

PC HID Tester

Due to time constraints, a HID tester application from the open-source community was used to test the device. It utilises standard Windows API calls to obtain a handle on the USB device by specifying the device's `VendorID` and `ProductID`, which are defined in the device descriptor of the firmware, see *5.4 Example: The Device Descriptor* on page 38.

Once a handle is obtained, the application can send and receive reports with the device. The report data the device sends is generated either by the GBA or randomly, depending on the state of a DIP switch (labelled `SW4` in *Illustration 4.3.3 - Master schematic for project hardware* on page 28). The report data consists of two single-byte axis and a byte button map, hence the ASCII transmissions of the GBA are converted to relative-coordinate axis data[35] by the P877 before transmission.

This application effectively demonstrates USB↔Device communication, and hence USB ↔GBA communication in both directions.

7.2 - Project in Review

The system created during this project ultimately was a success, and the aims of the project remained surprisingly static during its course, with the resulting system complying with the original specification.

However I do not consider the system to be complete, as standards-compliance testing could not be undertaken in any form. Official standards-compliance testing requires that the developer be a registered USB-IF member, and that they pay a fee; successful products result in certification and permission to use the USB logo on the product. The USB-IF recommend using a tool they provide, the *Command Verifier* application[36] to test devices before applying for official testing. I found this application did not function correctly, as it would not recognise *any* attached devices.

During all development, a test-first approach was used to discern which requests and firmware functions to implement next. Development was concentrated on the success-path of the system, and while I am confident that the systems fails specific requests in a compliant manner, I am not certain the system, when treated as a whole, is compliant. There may be requests required by other

35 The report descriptor defined each axis to use relative-position coordinate data, as opposed to absolute-position.
36 See *4.2 USB Command Verifier* on page 23

operating systems (such as *Mac OS X*) which are part of the standard, but not correctly implemented in the firmware. As mentioned earlier, this is the case with Windows 98 Gold and the HID Get Report request which it uses, requiring use of the control pipe to transfers reports, instead of the usual interrupt endpoint. As no thorough testing with other systems was undertaken I can only be sure the device is compatible with Windows XP.

As in USB itself, the device is a slave to the host, and hence the GBA is a slave to the PC application. Transmissions from the GBA are buffered in the P877 until the host polls for them. It is up to the PC software to make best use of the GBA to perform its intended task. The current demonstration applications have no intelligence, meaning that, for instance, the GBA does not receive confirmation from the P877 that a character it sent was received and there is no flow control.

Development of the PC aspects of this project were limited by time, and relatively little understanding of how an application interacts with a device was obtained. For instance, it is not clear how the native HID class driver of Windows handles interrupt data it acquired when no application is sinking it, as is the case when using the HID-Joystick firmware when not running the HID Tester application or viewing the data in the Game Controllers panel of Control Panel.

If I had to do this project again, I would reconsider my choice of USB interface chip. During this project I discovered a USB “solution chip” family called EZ-USB, manufactured by Cypress[37].

This family of chips have an integrated micro-controller. An EZ-USB does not feature static memory, but rather needs to be loaded with a firmware upon attachment to a host PC. When attached to a host, the device enumerates itself as an EZ-USB device and a tool supplied by Cypress is used to load the device with firmware – this can be automated (as in the case of commercial deployment). The tool, `fxload`, has a Linux version as well.

7.3 - Limitations

The firmware contains some known limitations, which were not addressed for various reasons, and the project itself was limited in some respects. The major limitations are summarised below.

Large (multi-transaction) Reports are not supported

HID Report-structures defined to be larger than the PDIUSBD12's Main OUT endpoint buffer (64 bytes) are not supported: the firmware is unable to send or receive such reports.

37 `http://www.cypress.com`

An internal buffer on the P877 large enough to accommodate such reports cannot be allocated due to RAM limitations. Code is in place to deal with such requests, adapted from the algorithm used for transmitting descriptor data over the (much smaller) control endpoint buffers, which is core to the enumeration process and known to work with multi-transaction data (as used during device enumeration). However, it was not possible to adequately test this adapted implementation.

By increasing the amount of available memory (perhaps by attaching an external RAM module, or using a higher-specification micro-controller) this could be over come.

HID Get Report request has limited support

It would be more correct to say that my implementation of the Get Report HID request (as described when mentioning Windows 98 earlier in this chapter) could not be adequately tested. To reiterate was was said earlier: Windows 98's HID class driver implements and old version of the HID class specification, which did not permit interrupt endpoints, so all report data would be acquired by the host using a Get Report request. Both old and current versions of the HID class specification only permit report data to be sent to device via the control pipe and a Set Report request.

Set Report requests are known to work (as data sent from the HID Tester application appears on the GBA) but Get Report was difficult to test. I did not have access to a Windows 98 Gold system to try it, and instead attempted to use the HID Tester application's “Use Control Transfers Only” option to test this. Usually this application will poll the interrupt endpoint, and send a Set Report request over the control endpoint when the user instructs it to send and receive data, but when “Use Control Transfers Only” is enabled it will emit both a Set Report request and Get Report request simultaneously. The Set Report request completes successfully in both cases, but Get Report returns null data of the expected length.

It is unclear if this is caused by a firmware conflict with Set Report (as it is likely that if the Get Report request is issued first, that the arrival of a Set Report request will cause the firmware to drop the former), or if this is a glitch in the HID tester. I believe that the HID tester is reliable, as it is used by many developers – the author (Jan Axelson, author of *USB Complete*[2]) and her website[38] are both reputable in the USB community (this is clear in the USB-IF Developers Discussion Forum[39]) – and so it is likely to have had plenty of feedback if it was faulty.

I believe checking the details of how Set/Get Report requests are handled and checking for overlapping use of buffers and such would be the best course of action to correct this limitation.

38 `http://www.lvr.com/usb`
39 `http://www.usb.org/phpbb/`

However, it is worth noting that Get Report requests are extremely rare, and not used at all in Windows XP which favours interrupt transfers, as per the recommendation of the current HID class specification.

Only tested with Windows XP

No attempt was made to test or develop this implementation of USB with any system other than Windows XP. When I conceived this project I knew nothing about how windows managed its peripherals nor how to access them from a user-mode application (nor did I know the distinction of user-mode and system-mode).

Therefore I feel an approach similar to my own undertakings might be successful: Using the debugging information from the device to discern what functionality and requests are lacking in the firmware and implementing them.

The device is not suitable for bulk transfer of arbitrary data

The firmware created for the device implements a HID. All data transfers with HID class devices must be done through reports, which are structured. It is therefore not suited to transmitting, say, files, as these will render the report structure irrelevant while restricting the size of data packets as well as requiring the PC software to also disregard the structure and reassemble the separate reports.

Furthermore, HID class devices use interrupt transfers for IN data, and the control pipe for OUT data, neither of which are suitable for bulk data transfers. The correct transfer type to use would be the bulk transfer mode that USB supports. This is discussed in slightly more depth in on page .

Power consumption regulations not considered

USB has strict regulations on power consumption for bus-powered devices, for instance a device may only draw a maximum of 100mA until it is in the Configured state and the host has agreed to supply it with more (up to a maximum of 500mA). Devices are required to support a “suspend” state, where their power consumption must not exceed 500μA.

USB devices must enter the suspend state if they do not see a Start-of-Frame marker on the bus for at least 3ms. The PDIUSBD12 triggers an interrupt (a 'suspend change' interrupt) when this occurs.

The device created in this project was bus-powered for convenience, but does not support a suspend state, instead the PDIUSBD12's suspend functionality is disabled by tying its `SUSPEND` pin low. This did not appear to cause any problems during my tests, while it definitely breaks the USB specification, Windows appears to be quite forgiving in this respect.

Both the PIC16F877 and PDISUBD12 feature a low-power state which may well provide an answer for this, the PDIUSBD12 data sheet suggests that the PDIUSBD12 can be woken by either the micro-controller or USB bus it is attached to, and will trigger and interrupt for the latter.

7.4 - Recommendations for Further Work

As discussed in the introduction of this report, I intend to use the work I have done in this project to create my own PDA application for my GBA, doing so should be possible by tweaking the existing implementation, however during the course of this project a few ideas of new directions the work could be used in occurred to me.

Dynamic Sensors

While learning of the HID report structure, and its ability to associate units of measurement with data fields, it occurred to me that a generic monitoring application for the host PC could be created, which would parse the report descriptor and setup a display of the data with associated units, as well as providing data logging and visualisation based on the units.

This could provide a convenient interface to connect multiple devices each monitoring different things, and have them dynamically allocated screen-space (or equivalent) to visualise the data, or in a more complex system were all readings relate to the same model, allow further refinement of simulations.

Specifically it occurred to me that such a facility would be useful in creating mobile robots, as sensors connected via USB could have associated units and scales specified in their report descriptor(s). This could ultimately lead to a pseudo-generic mobile robot for which individuals could construct complex sensors, each able to process readings to produce other readings (such as is done in triangulating position from sonar data).

Implementing a USB class other than HID

The HID class implementation used was added on top of a generic USB firmware (which enumerates the device), and involved relatively little change – the addition of two new report-related requests and the use of an interrupt endpoint.

Hence it should be relatively simple to convert the firmware to implement a class other than HID, the one that springs to my mind is the Mass Storage Device class, as used by USB pen-drives and portable hard disks.

Having briefly looked into what the Mass Storage Device class entails, I see that the class itself supports two distinct transport protocols (sub-classes): Bulk-only and “Control/Bulk/Interrupt”

(CBI). Both implement very few class-requests, two for bulk-only and just a single “Accept Device Specific Command” request for CBI. A descriptor defines what set of command blocks the device actually uses to manage its data.

The current Mass Storage Device specification supports 5 command blocks ranging from “Reduced Block Commands (RBC)” which is typically used in Flash devices, to “SCSI transparent command set”.

It appears that the USB side of the implementation would be minimal, the focus being on implementing a command block appropriate to the storage medium used.

8 - About the attached CD

8.1 - Game Boy Advance Applications

The GBA applications developed while working on this project are included on the attached CD, and are located in "`/Game Boy Advance`" and are further divided into directories by application. Each application contains the necessary source code to compile the application in the HAM environment, a pre-compiled binary suitable for multi-boot[40] and a file `INSTRUCTIONS.TXT` which serves to explain the application's purpose and it's usage.

VisualBoy Advance, a GBA emulator for Windows, has been included for convenience; however it is of limited use as it does not emulate the communication port.

HAM is available from `http://www.ngine.com` and a version is included on the disc for your convenience. Please be sure to check the website for the most up-to-date version.

8.2 - Copies of Electronic References & Resources

Copies of significant electronic references and resources have been included on the CD to ensure that the reader can view the same content as the author. The transitive nature of the web tends to make internet-based references of little use as they offer no traceability, however much of my project was based around the GBA home-brew community which only exists on such a medium.

The references are located in `/Reference Copies` and other resources (such as a copy of the HAM development environment for the GBA) are located in `/Resources`.

8.3 - Firmware source code (including examples from Philips)

All example source code obtained from Philips and all code developed personally is located on the CD in `/Firmware Source Code`. The Philips code was obtained from *Philips Semiconductors Download Area* which I gained access to by registering an account on `http://www.semiconductors.philips.com/`.

The developed source code includes snapshots of the firmware at what I considered to be mile-stones in development: getting the GoodLink LED to light, getting the device to enumerate, enumerating without using RAM to store descriptors in, a HID-mouse implementation (combining GBA control) and finally a HID-joystick implementation (again using GBA control).

40 That is to say, it works when sent over as bootstrap code

9 - Acknowledgements

I would like to thank Dr. Roger Packwood for his invaluable advice, open approach and teaching me to always tackle the highest-risk first in my work. I also wish to thank Mr Rod Moore for his advice and enthusiasm about this project, it is satisfying to know that this work may be used by others. Finally, I would like to thank Mr Barry Thorne, Mr Franc Buxton and Mr Stuart Valentine for welcoming me to their laboratory and creating a friendly atmosphere where I quickly felt at home.

I wish to formally acknowledge the work of Jan Axelson, author of *USB Complete[2]* which served as my key reference throughout the course of this project.

I also wish to formally acknowledge the work of Craig Peacock, of Flinders University, Australia, who is the maintainer for BeyondLogic.org and its *USB in a NutShell[3]* multi-part article, around which the ground-work of this project was based.

10 - References

[1] *Nintendo - Customer Service -- Corporate Info - Company History*, viewed Wed, 6 Apr 2005, `http://www.nintendo.com/corp/history.jsp` (on attached CD)

[2] Jan Axelson, *USB Complete: Everything You Need to Develop Custom USB Peripherals*, Lakeview Research, 2001. ISBN 0-965082-990

[3] *USB in a NutShell*, viewed Tue, 12 Apr 2005, `http://www.beyondlogic.org/usbnutshell/usb7.htm` (on attached CD)

[4] *Gameboy Advance Technical Info (GBATEK)*, viewed Wed, 6 Apr 2005, `http://www.work.de/nocash/gbatek.htm` (on attached CD)

[5] *DarkFader.net Game Boy Advance Development*, viewed Thui, 14 Apr 2005, `darkfader.net/gba/main.html#GbaTool` (on attached CD)

[6] *FiveMouse.com GBA Web Server*, viewed Thu, 14 Apr 2005, `http://www.fivemouse.com/gba/` (on attached CD)

[7] *PDIUSBD12: USB interface device with parallel bus*, Koninklijke Philips Electronics N.V., 2001 (on attached CD)

[8] *PIC16F87x: 28/40-Pin 8-Bit CMOS FLASH Microcontrollers*, Microchip Technology Inc., 2001 (on attached CD)

[9] *Firmware Programming Guide for the PDIUSBD12*, Koninklijke Philips Electronics N.V., (on attached CD)

[10] *Philips PDIUSBD12 FAQ*, viewed Mon, 18 Apr 2005, `http://www.semiconductors.philips.com/cgi-bin/faq/`

11: Robert Meerman, Project Spec: Making a Personal Digital Assistant on a Game Boy Advance, (see Appendix), 2004

11 - Appendices

11.1 - Project Specification

Making a Personal Digital Assistant on a Game Boy Advance

Problem

Powerful Personal Digital Assistants (PDAs) and smart-phones are becoming common place and are increasingly used to entertain their users with games. This is fine for those that predominately want an organiser and think of games as a bonus, but what about those that prefer the games, but would still like some organisational capability? Portable games consoles are powerful enough to run simple PDA-like applications but they lack the connectivity that PDAs and smart-phones have in abundance.

Objectives

To create an interface between a PC and a Nintendo Game Boy Advance (GBA) system which is as easy to use as any existing PDA and to create some proof-of-concept software that shows the use of this interface to communicate between the two.

Breakdown

- A GBA↔PC interface which requires no special drivers to be installed on the PC or GBA, and which in the presence of suitable software, communicates effectively between the two devices.
- A proof-of-concept application pair which demonstrate the effectiveness of the solution.

If time allows:

- The applications will be expanded to work with real data, such as the to-do lists / contact list of larger applications such as MS Outlook using SyncML.

Methods

Pre-requisites phase

Reading up on the subject and evaluating options before settling on a path.

- Learning how to effectively program the multi-player port on the GBA hardware and choosing the more appropriate mode to use.
- Choosing and evaluating the PC port to use for interfacing. Currently favoured is USB for its wide acceptance and easy-to-install end-user products.
- Choosing a micro-controller to interface between the two devices, which will be used to translate signals encapsulated in a standard-compliant protocol from the PC to a (simple) protocol of my own devising on the GBA.

Maturing phase

Becoming acquainted with the hardware:

- Probing the GBA hardware and testing multi-player port controlling code while determining the properties of the signals (voltage etc)

- Probing the chosen PC interface and testing code which manipulates this while determining the properties of the signals (voltage etc.)

Once the results from the above are predictable and well understood:

- Create some signal translation circuits as applicable (perhaps the voltages are different for each device, that would be taken care of here)
- Probing / coding the micro-controller with some test data and getting a feel for the capabilities of the device (response times, speed, caveats)

Implementation phase

Bring the components together:

- Assemble the interface made up of the GBA connector cable, PC connector cable and micro-controller.
- Set-up micro-controller to be identified and used by the PC (as applicable to the chosen PC port – i.e. USB will need PnP information, protocol setup etc.).
- NB that the GBA does not require special set-up as it does not have an operating system. Instead it is only required that the multi-player port chip-set is in the correct mode (chosen during pre-requisites phase)

Create some proof-of-concept software which makes use of the new connectivity of the GBA such as synchronising a to-do list, contact list or similar task.

- Code an application for the GBA which displays incoming data on-screen.
- Code an application for the PC which generates test data and sends it to the GBA.

Once this is working:

- Code applications to do the reverse (PC -> GBA)
- Combine these into single bi-directional applications which swap test-data.

If time allows:

- The protocol of the GBA will be improved to resemble a more orthodox protocol.

Timetable

Checkpoint	*Description*
End of week 5	Chosen PC port
End of week 6	Finished reading up on GBA multilayer port, experimentation/probing started Chosen micro-controller
End of week 7	Experiments / probing PC port under way
End of week 8	Start designing/profile needs of translation circuits
End of week 9	Micro-controller experiments under way
End of week 2 Term 2	Interface able to (partially) transmit / receive data
End of week 4 Term 2	Interface stable and working Start development of applications
End of week 6 Term 2	Minimal applications complete Test on other PCs (non-development ones) for installation issues and compatibility
Mid Easter break	Project report is in draft essay form which contains all relevant information, but may been reworking.

Thurs week 2, term 3 Submit Project report.

Resources

The following will be required in addition to those currently available to me within the department (such as the Linux workstations):

- Direct access to the parallel port on at least one workstation (for programming the GBA hardware using my personal 3rd-party programmers)
- Direct access to the serial port on at least one workstation (for debugging the GBA hardware in real-time using my personal 3rd-party debugger)
- Frequent access to a PIC-programmer (for the interface micro-controller)
- Manuals (possibly online) which explain how to use the PIC-programmer
- Manuals / reference on programming PICs (dependant on the brand of PIC chosen, it is expected that the project supervisor will guide this choice to best suit the resources available)

11.2 - Parts list for Master Schematic

Item	Part Value	Description	Qty	Parts
1	10uF	Electrolytic Capacitor	1	C1
2	0.1uF	Electrolytic Capacitor	5	C2 C3 C4 C5 C6
3		Light Emitting Diode	3	D1 D3 D4
4		Push button normally closed	1	PB1
5	1MΩ	Resistor	1	R2
6	120Ω	Resistor	4	R3 R4 R5 R6
7	4.7kΩ	Resistor	3	R1 R7 R8
8		Single pole switch	4	SW1 SW2 SW3 SW4
9	PDIUSBD12	USB Interface Device	1	
10	PIC16F877	Micro-controller	1	
11	MAX3232	RS232 level translator	1	
12	4MHz	Ceramic Resonator	1	X1
13	6MHz	Crystal	1	X2
14		Component Header	2	GBA_Header USB_Header