University of Salford
A Greater Manchester University

School of Computing, Science and Engineering
Acoustics, Audio & Video Final Year Project

This paper represents my own work. Any input or work done by other people is clearly noted and properly referenced.

Generating MIDI data and the application of modern software design techniques

David Lewthwaite

@00113245

d.lewthwaite1@student.salford.ac.uk

MIDI has always been traditionally controlled via the musical keyboard style interface. In terms of musical expression it is far from ideal. This system makes use of the features within a Nintendo WiiMote to accurately simulate a musical instrument in the form of a drum-kit. No other WiiMote derived system uses the physical position of the users' hands to determine which pitch or drum to trigger. In the modern world, software usually forms one of the key components to a product. Therefore, modern software design techniques are employed to create the applications and interfaces required for connecting MIDI to a new physical device.

Contents

Generating MIDI data and the application of modern software design techniques 1
Contents 2
List of Figures 5
List of Tables 6
Chapter 1. Background 7
1. Introduction 7
1.1. Current Research 7
1.2. The Project 8
Chapter 2. MIDI Communication 9
2. MIDI DLL 9
2.1. Windows API 9
2.2. Design Approach 9
2.2.1. Modularity 11
2.2.2. DLLs & Libraries 11
2.3. Cross Compiler Compatibility 14
2.4. Test Application 16
2.5. Improvements 17
Chapter 3. WiiMote Communication 18
3. WiiMote Communication 18
3.1. The WiiMote 18
3.2. Research Issues 18
3.3. Interfacing in C++ 19

3.4. WiiMote Messages 20
3.5. Data Sources 21
3.5.1. Buttons 21
3.5.2. Accelerometer 22
3.5.3. IR Sensor 24
3.6. Use of the data 26
3.6.1. IR for On-Screen Positioning 26
3.6.2. Accelerometer for Velocity 27
3.7. Algorithms 27
3.7.1. IR Trigger Algorithm 27
3.7.2. IR Direction Sensing Algorithm 29
Chapter 4. System Design 31
4. Software Design 31
4.1. Prototyping to Final 31
4.2. Software Structure 32
4.3. System Lifecycle 35
4.4. Modular Design 37
Chapter 5. Test Systems 38
5. Software Testing 38
5.1. Technical Testing 38
5.2. User Testing 41
5.2.1. Software 41
5.2.2. The Tests 43
5.2.3. Analysing the Results 44

5.2.4. Results Usability 46

Chapter 6. Conclusions & Future work 47

6. Conclusions 47

6.1. Further Work 48

7. Acknowledgements 48

8. References 48

9. Appendix 53

9.1. Project Plan 53

9.2. Explanation of Bit Shifting and OR Operator 56

9.3. Explanation of Class Inheritance 57

9.4. ASCII Chart 58

9.5. Explanation of Bit-Masking and Bitwise AND Operator 59

9.6. Explanation of Function Pointers in Borland 60

9.7. User Testing Questionnaire 62

9.8. User Testing Results Tables 64

9.9. Accompanying CD-ROM 68

List of Figures

Figure 1: Flow diagram illustrating the life cycle of the MIDI Interface API............................. 13

Figure 2: Screen-shot of the demo application. 16

Figure 3: WiiMote axis reference [15].. 23

Figure 4: Trigger algorithm operation 28

Figure 5: Direction sensing algorithm operation 30

Figure 6, Development application... 32

Figure 7: Application structure 33

Figure 8: WiiMote window.. 34

Figure 9: Software lifecycle.. 36

Figure 10: DUnit test for MidiInterface... 39

Figure 11: Demo application main screen 41

Figure 12: Results graph of all users per pattern 46

Figure 13, Project plan timeline... 55

Figure 14: ASCII table [20] 58

Figure 15: Callback example 60

List of Tables

Table 1: MIDI Short Message Structure 10

Table 2: Bit packing of WiiMote button status 21

Table 3: Typical raw values for acceleration in each dimension 23

Table 4: Report 0x33 Structure 25

Table 5: Example results table 45

Table 6: Test subject 1 results 64

Table 7: Test subject 2 results 64

Table 8: Test subject 3 results 65

Table 9: Test subject 4 results 65

Table 10: Test subject 5 results 66

Table 11: Test subject 6 results 66

Table 12: Test subject 7 results 67

CHAPTER 1. BACKGROUND

1. INTRODUCTION

MIDI, or Musical Instrument Digital Interface, has been around now for well over 20 years, its adoption by the audio and video industry has been ubiquitous with MIDI being found in many pieces of equipment, from synthesizers to control surfaces and lighting desks. However, generating MIDI data from user input has almost always been confined to a MIDI keyboard. Other devices do exist such as MIDI-guitar interfaces [1], woodwind instruments [2] and generic controllers typically linked to synthesizers and digital audio workstations (DAWs) but these never gained widespread use.

This project covers how MIDI communication is implemented within modern software as well as using new physical devices, which may not have been designed for the task, to make music. Modern software design focuses on modularity and re-use, throughout the course of this project, industry accepted practices and ideas have been employed to create modules which pave the way to future projects and development.

In order to avoid repeating previous work, much research was carried out into previous and current products and research. This served as a means to provide information about which areas where there was a lot of information and those which suited an extension to the research which already exists. Section 1.1 details current and previous work.

1.1. Current Research

Preliminary research indicates that while there have been a number of new user interfaces for MIDI, such as the Alesis AirDrums [3], none have been as accessible or inexpensive as re-purposing a device originally intended for computer games. An example of this is the Nintendo WiiMote. The WiiMote is the controller for the Nintendo Wii, but its feature set is far beyond anything in the same price range of around £30. The WiiMote features,

- 3-axis Accelerometer
- High-Resolution IR object tracker
- Bluetooth Connectivity

Not to mention the basic attributes such as buttons and vibration motor. Many people have been able to use the WiiMote for fun and interesting tasks such as tracking the motion of a conductor conducting an orchestra [4]. In commercial applications, Nintendo themselves produce Wii Music which allows the user to play many different musical instruments virtually [5]. The method of implementation is quite simple – the accelerometer acts as a trigger for notes with the buttons representing different pitches. In April 2009 a new game was released, ‘We Rock: Drum King’ [6]. This is a drumming game which also uses physical movement to trigger the drums, but

button presses to distinguish which drum sound to use. In a more sophisticated system, the user would be tracked in 3D space to allow for a more realistic simulation.

Of course there are other areas of research in relation to this field, much work has gone into producing computer interfaces which can represent MIDI controls or effects graphically [7], where a user can draw with the mouse cursor, or other device onto axes which are mapped to MIDI controls, in fact, some people are developing software which allow the users to load in photographs or images to be processed in the same way.

A further field which has been pushed to the forefront of recent research is neural control. Some companies are working on systems which allow users to play games with the power of their mind. One system is the OCZ Neural Impulse Actuator which uses a series of sensors to measure brain activity in alpha and beta channels as well as muscle and eye movement [8]. These kinds of devices could easily be applied to music and MIDI data generation.

1.2. The Project

This project has developed a new C++ library for interfacing with MIDI devices through Windows as well as a method of interacting with a new device to generate MIDI data. Since the WiiMote provides the greatest scope for an interesting development, this has been the focus of the data source.

MIDI has been around so long now that the original methods of accessing MIDI devices in the Windows operating system are beginning to be outdated. The project has addressed this issue to bring it in line with modern software practices and has taken advantage of the features the relatively new language of C++ provides.

Of all potential alternative input devices, barring the PC mouse, the WiiMote is probably one of the most common with over 20 million Wii consoles sold by the beginning of 2008 [9]. The device interfaces over Bluetooth and does not require any special cables or adaptors. The project plan as submitted in November 2008 is at Appendix 9.1. Where the project deviated from the plan is discussed later.

CHAPTER 2. MIDI COMMUNICATION

2. MIDI DLL

One of the requirements for this project was to produce MIDI data from an alternate input source. This requires a software MIDI interface. Although on the Windows platform, Microsoft provides an API, or Application Programming Interface, for Multimedia communication, it is relatively complex and the methods it employs in its design are now considered quite archaic. An API is a set of defined functions designed to perform specific tasks or sets of tasks, in the case of the Windows Multimedia API these cover functions for audio, MIDI and games controllers.

2.1. Windows API

The Windows API for MIDI consists of a collection of C functions, the most important of which is `midiOutShortMessage()`. This function is how all MIDI messages (apart from SysEx) are sent. However, the messaged is fully encapsulated into a single 32-Bit value. Having to encode the message into this format every time is quite tedious and creates unnecessary additional coding problems. It is also quite easy to incorrectly encode.

2.2. Design Approach

In order to ease the use of MIDI within Windows, a C++ class interface can be used to bridge the gap between a developer's application and the Microsoft Multimedia API. A class can be thought of as a 'module'. These 'modules' generally have a single use with well documented functions and data structures, here the class will be for performing MIDI interface operations. Classes are a very useful way of designing software as it allows developers to update and replace classes easily later. Using modern C++ class design techniques, the calls are encapsulated into: creation, enumeration and initialisation, data transmission and destruction methods.

Instead of manually enumerating through the MIDI devices using `midiOutGetDevCaps()`, the developer can now create an instance of the `MidiInterface()` class using the `new` operator. From this the developer can access a full valid device list and choose which to initialise with. This internally opens the device handle to the MIDI interface and the developer can then access the appropriate method with valid data to perform the desired action.

For example, the note on command consists of the following:

```
midiOutShortMessage(HANDLE MidiDev, DWORD Msg);
```

On the face of this, it looks like a difficult to understand function; however, the `DWORD Msg` contains a 32-Bit value which contains the 3-byte short MIDI message. There are four major parts to this value, it has the following structure

Bit	Use
1-4	4-Bit Status Value
5-8	4-Bit Channel Number
9-16	First Data Byte
17-24	Second Data Byte
25-32	Unused

Table 1: MIDI Short Message Structure [10]

In order to make this functionality easier to use, the new design includes a function dedicated to encapsulating the data, called `PackShortMessage()`. This first copies all the input data into new 32-Bit values, then uses the bit shifting operators (>> and <<) to order each part of the message to the correct place within its own 32-Bit value. Combining these into a single value is as easy as adding them together. See Appendix 9.2 for an explanation on Bit Shifting and the bitwise OR operator.

This means that instead of having one 'message send' function, there are clearly defined separate functions for sending MIDI data. To send a NoteOn message, the developer calls,

```
int __stdcall MidiInterface::NoteOn(unsigned short aNote, unsigned short
aVelocity, unsigned short aChannel);
```

This is significantly more self-explanatory as to its use than the default Windows API. All of the standard short message functions have error checking on the input values. Should any of the parameters be out of range, the function returns with the value of -1 so that the developer knows there as been an error.

The `MidiInterface()` class contains an error decode function whereby if one of the Windows API calls fails, it will decode the error value into something more meaningful. At present this error is not used, but during development, the developer can place breakpoints at the end of this function to determine which error occurred.

Dealing with SysEx data requires a different approach, since this can be of arbitrary length and contain almost any sequence of values, all that is defined is a memory pointer to the data and a length value. The developer must '`new`' the data block to be used for the message, fill out appropriately using the standards defined structure – byte 1 is `0xF0`, byte 2 is the manufacturer and device ID followed by an arbitrary number of bytes for the message. A SysEx message is always finished with a value of `0xF7`. With the exception of the beginning and end bytes, all

the values range from 0x00 to 0xFF. The SysEx function will automatically 'delete' and clean up the memory allocated for the message.

2.2.1. Modularity

This object orientated approach to the design of a new MIDI interfacing class allows for future modular style development where another developer will create a new class which inherits from this MIDI interface. This new class could cover a specific derivation of MIDI such as XG or manufacturer such as Yamaha. It could include direct functions which are appropriate to that device such as having a function to directly set reverb parameters which automatically fills out the correct SysEx message and passes it to the SysEx function in the base MIDI interface class.

Someone who wishes to implement a Yamaha SW1000 interface could then simply create a new instance of the inherited class and immediately have access to all of the appropriate functions. See Appendix 9.3 for an explanation on class inheritance.

2.2.2. DLLs & Libraries

In order to make this new MIDI communication class easy to use by other developers in their own applications, it is compiled as a DLL or Dynamic Link Library. When software is compiled, it consists of two main processes – compilation and linking.

In compilation, each separate class within the software is compiled into its own object file. These object files are then linked together to form an application. A library is simply a collection of objects in a single file. These are distributed with their associated header files which tells the compiler and developer which functions are available and how to use them. The library is then added to a developer's project in the same way as a source file and linked in at compile time. A dynamic link library is distributed with both a header file and a 'skinny' library file. The library file tells the linker that whenever the software is actually executed, it should load up the DLL file which contains the object files.

A DLL allows for post-compile updates to be applied. This is especially useful when the DLL has been produced by a third party vendor and so the developer is not responsible for software updates. It is also useful in large projects as it prevents the main application executable from becoming large in file size.

To export a C++ class as a DLL in the main chosen build environment, Borland Codegear C++ Builder, it is necessary to add `__declspec(dllexport)` within the class definition of the header file. The compiler then interprets this and inserts the functions into the DLL. The header file distributed with the DLL must differ slightly from this in order to tell the compiler of the new application how to use it, by defining the class with `__declspec(dllimport)`.

Using the class is remarkably simple. All the developer needs to execute is the following

```
MidiInterface* Iface = new MidiInterface();
```

Internally this calls the `Enumerate()` method and populates the device information structures which can be accessed through

`Iface->InDevList` and `Iface->OutDevList`

The structures have the following format,

```
typedef struct
{
        unsigned char DeviceName[MAXPNAMELEN];
        unsigned short DeviceID;
} MidiDevice;
```

The total number of input and output devices are stored in integer values at

```
Iface->MIDIInDevs

Iface->MidiOutDevs
```

The developer can then iterate or display the devices to the user before calling

```
Iface->Initialise(MidiDevice *aInDev, MidiDevice *aOutDev);
```

The initialise function will take a pointer to one of the MIDI devices and open the physical hardware ready for communication. This only needs to be executed once before using the MIDI functions. If the initialise function is called again, the class will automatically close up any open devices and open the newly selected ones. Automatically closing devices on re-initialisation makes the application more stable as it doesn't leave a device in a 'waiting' state or any leaked memory, both of which can cause an application to crash.

After these initial setup steps, the developer can access any of the MIDI functions. Most of the General MIDI (GM) set is implemented including RPNs, NRPNs and SysEx. The function definitions in the header file (MidiInterface.h) were written with clarity in mind and should be self explanatory as which parameter refers to which part of the MIDI message.

When the developer is finished with the MIDI Interface, calling the delete operator is all that is required i.e.

```
delete Iface;
```

Figure 1 illustrates the lifetime of the `MidiInterface()` class. As is evident, the class is quite linear in its operation only having one path back to the beginning when the developer chooses to change the input or output device in use.

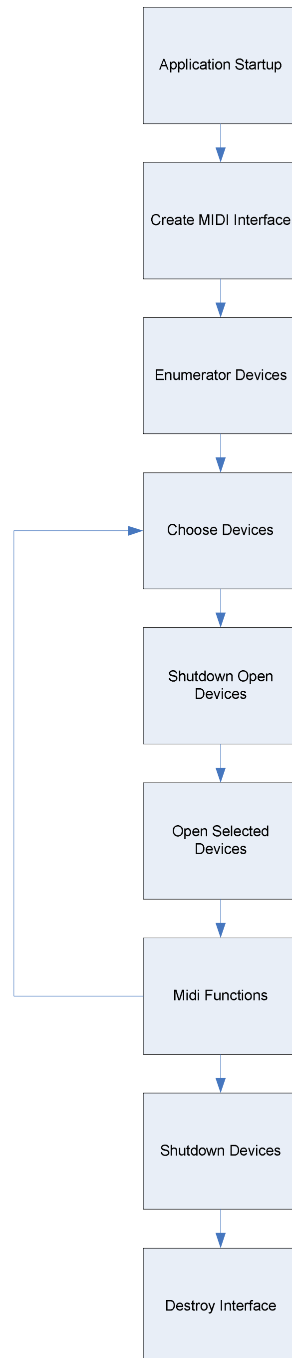


Figure 1: Flow diagram illustrating the life cycle of the MIDI Interface API.

2.3. Cross Compiler Compatibility

Although the main development language used to create the MIDI interface library was Borland Codegear C++, it is recognised that other languages and variants are also widely used by both commercial and open source developers. On the Microsoft Windows platform, these are often Microsoft Visual C++, Visual Basic and Visual C#. In order to ensure cross compiler usability, standard C++ programming techniques are employed, implementing only those features which are in the C++ coding standard or which are present across the platforms.

In porting the library from Borland C++ into a more standardised format, the first thing which needed to change was the calling convention. A calling convention is placed within a function definition as a way to tell the compiler special instructions on how to compile that function, often as a way to optimise the code when it is executed. For example, the following function has a defined calling convention of `__fastcall`, has no arguments and no return type.

```
void __fastcall ClassName::FunctionName( void)
```

By default, Borland uses the `__fastcall` calling convention, which specifies that any arguments given in the function are passed in CPU registers wherever possible [11]. This is a speed performance optimisation as data is not automatically placed straight into main memory which takes longer to access.

In order to achieve cross compiler compatibility, a calling convention which works across both Borland and Microsoft Visual C++ is required. Microsoft state that the convention of `__stdcall` should be used when calling Win32 API functions, of which the Multimedia API is a subset. `__stdcall` ensures that data passed between developer applications and the Win32 APIs is correct. This is a more stable (although not optimised) calling convention. To this end, the MIDI interface library was re-written to use `__stdcall`.

As well as the requirement to change the calling convention used within the library, the Microsoft Visual C++ compiler is stricter in regards to forcing function definitions to be correct. For example, where Borland would produce a warning but continue, the Microsoft compiler will error and force the developer to fix the issue. These issues generally stem from functions having return types specified but which did not actually return anything, in which case, the type '`void`' is more appropriate. As a consequence of this, the code in the Midi Interface library is now of a high strict standard and will compile and execute across both compilers without issue.

Although the library can be compiled on both platforms, the compiled library or DLL cannot simply be interchanged between the two. Because of the way C++ code is compiled together, the linker needs to be able to reference each function within an application in a unique manner. In written code, this is obvious as all C++ functions must be within a class definition, however, when the compiler converts these to machine code, it must have a way of compacting down the long human readable names. This is called name mangling and unfortunately, C++ has one of the

least standardised techniques for achieving this. This leads to problems when importing libraries written in Visual C++ for use within Borland C++.

It is worth noting that in pure C libraries, which are not object orientated and so do not have classes, the name mangling is more standardised, or at very least is understood.

Not all developers use C++ however, and it is sometimes a requirement for other people to be able to use languages such as Visual Basic and Visual C#. Both of these have mechanisms for importing libraries; however, they suffer from not being able to use C++ objects. One way around this issue is to write a set of C style functions (which are not in classes) which 'wrap' each function within the C++ library so that when it is then exported, all the new application needs to access is a collection of C functions. It is worth noting that this technique also works for coding between versions of C++ as mentioned above as the new C library can be converted between Borland and Microsoft quite easily.

An alternate method of dealing with this issue is to code from straight C to begin with. However, given the inherent benefits of using a newer, object oriented language such as C++ this is often not really a desired approach, it does however, give the best compatibility between languages.

This leads onto why the choice of using C++ over C. C++ is a newer, more modern language, which although started development as an extension to C in the late 1970s, was not ratified by the ISO/IEC group until 1998 so is only now coming through in widespread use. This is mostly evident in the newer generation of Microsoft Windows programming APIs with the advent of COM and COM+ models. Older Microsoft APIs which have been around since Windows 3.1, 95 and 98 are presented in C, although these may be wrappers around the closed Windows source code.

C++ provides a mechanism of creating highly modular applications through the use of classes. Classes or objects mean that once a design is in place detailing how the different classes should interact – in terms of what data needs to pass between the two, different developers or groups of developers can write in isolation in full knowledge that as long as the design plan is adhered to, the classes should fit together. This means that classes can be updated, re-written and replaced later without huge implications for the rest of the system. This is most evident when writing for cross platform.

For example, on Windows, the Multimedia API provides access to MIDI functions allowing developers to communicate with MIDI hardware. The MIDI interface library uses these functions and so it would not compile on a Linux system as the API functions are different. However, it would be possible to produce a new MIDI interface library which has the same public function definitions as the Windows version, allowing the rest of the application to remain unchanged (if the rest of it is written to standards compliant C++).

2.4. Test Application

At the mid-point of development, there was a requirement to demonstrate the MIDI interface library in a working condition. Since presenting source code neither demonstrates if it works or is particularly interesting, a test application was produced to show each of the available functions. The test application was written in Microsoft Visual C++ and has the ability to send any type of MIDI message.

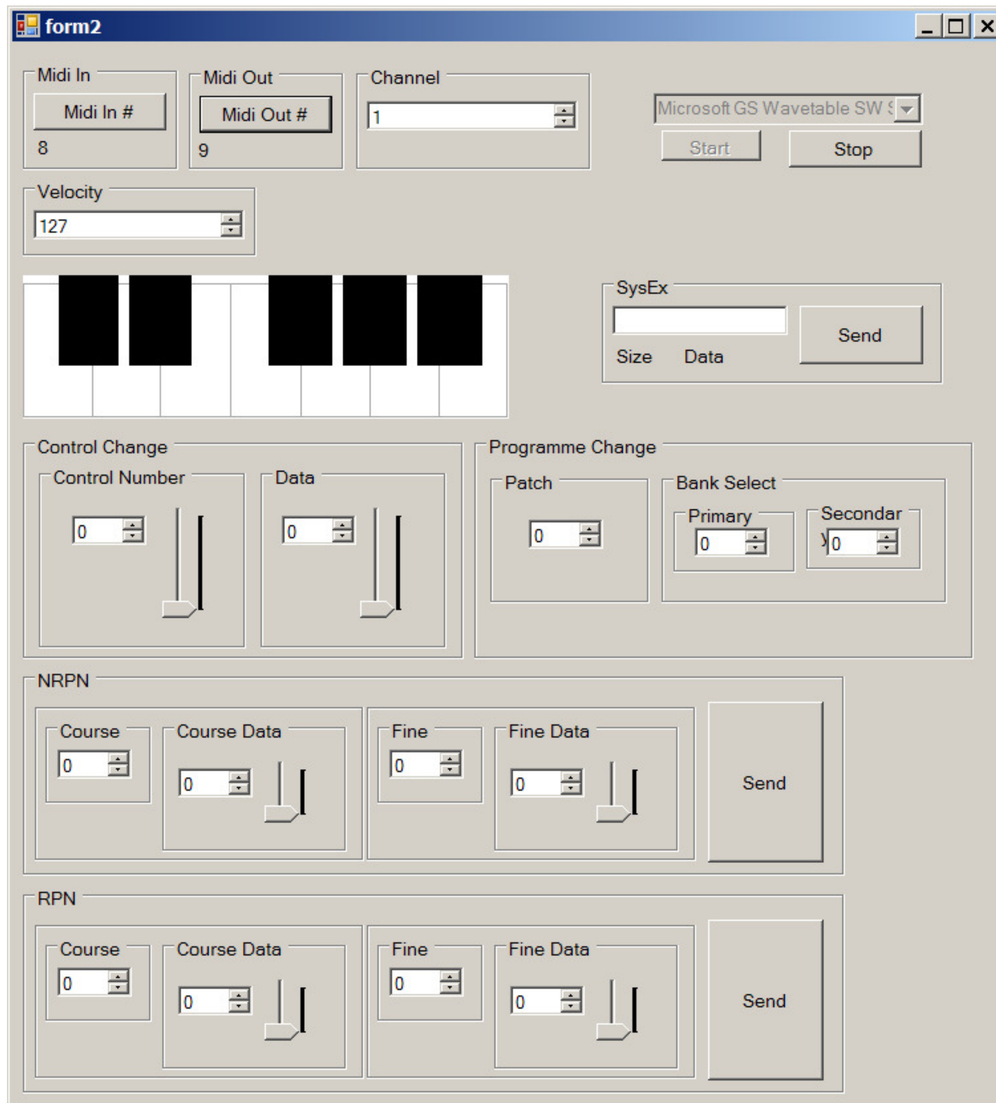


Figure 2: Screen-shot of the demo application.

Although, quite visually unappealing, the application does provide access to all of the functions in the MIDI interface library as it was at the time. The user would launch the application, choose a MIDI output device and click ‘start’. All of the sliders and values on screen then operate as

expected to control the relevant MIDI parameters. The mini on screen keyboard allows for an audible demonstration.

The event handlers of the sliders and buttons are connected to functions which use the values from the text boxes or sliders as parameters in MIDI interface library calls. The only part really worthy of specific mention is the SysEx functionality. Since SysEx is of arbitrary length, it is difficult to process. The input text is presented as a Microsoft system string, in order to convert this into an unmanaged array of char values, Microsoft provide some conversion routines to convert between the managed Visual C++ code and unmanaged pure C++.

The array of char values contains the alphabetic (American Standard Code for Information Interchange, or ASCII) representation of the values typed in by the user. ASCII values range from 0 to 127 in the standard set. However, the first part of the ASCII standard is used to represent commands traditionally used for modems and printers such as carriage return and new line. This means that the number '0' does not have an ASCII value of '0' instead it is represented by the value decimal '48'. See Appendix 9.4 for a copy of the ASCII chart.

This requires a conversion routine to convert from the ASCII values into their decimal equivalents. If decimal values 0-9 are represented by ASCII 48-57, it is trivial to say that for all values within the incoming char array which fall between 48 and 57, they can be subtracted by 48 to give their decimal equivalent. For example, if an incoming value is ASCII 57 (text character '9'), 57 – 48 = 9 which is correct.

Letters are dealt with in a similar manner, just the offset is different and so the two types of input are dealt with separately. Since each input character represents 4-bits, outputting the 8 bit numeric value from two 8 bit ASCII input values is simply a matter of bit shifting the first value up 4 places into a new 8 bit number and performing the `OR` operator with the second 4bit output value.

2.5. Improvements

Since the MIDI interface library was first created, a number of improvements have already been made. Most notably, a developer can now re-initialise the library with new choices of MIDI input and output without having to first shutdown. Although the library is now rather polished and usable, improvements still can be made. The most major feature which could be added is MIDI input functions – so the library can be used in applications which require two way communication. Multiple MIDI outputs are taken care of by the fact that multiple instances of the interface class can be created on different physical interfaces.

Additionally, if a developer wishes to take advantage of the openings created by the C++ design method – of using inheritable classes and objects, further, more device specific, interfaces can be produced which sit on top of the base MIDI interface and take care of encoding messages for manufacturer specific commands, such as SysEx for controlling reverb on a Yamaha SW1000 soundcard.

CHAPTER 3. WIIMOTE COMMUNICATION

3. WIIMOTE COMMUNICATION

The second major part of this project involved taking a physical device not intended for musical expression and deriving MIDI control data from it. The Nintendo Wii controller, or WiiMote, is an ideal device for such a task. Since its launch, the controller has generated a large interest in many different areas of research – from computer games and audio, to virtual reality and learning applications.

3.1. The WiiMote

The two main reasons the WiiMote has sparked such an interest are firstly, because it can be connected to a computer on almost any platform due to its design as a HID (human interface device) over Bluetooth and secondly because of its rich collection of sensors and data available.

The WiiMote consists of the following:

- Infra Red camera and tracker
- Accelerometer
- Buttons (12)
- Vibration Motor
- LEDs (4)
- Speaker
- Expansion Port

The primary focus here is to use the IR camera and the accelerometer; however, during research and development, it was trivial to implement control for the buttons, LEDs and vibration motor. The internet and academic community have been working hard for the last couple of years on communicating with the WiiMote and applying it to uses far beyond the intent of its creators.

3.2. Research Issues

Since Nintendo never intended for it to be used in the manner which so many people are, there were a number of issues with researching the WiiMote. The main problem is due to lack of available documentation and specifications. However, due to the nature of HID devices, the WiiMote must be self-describing as to the functions it has available via HID descriptors [12].

Some people have been able to use a Bluetooth packet sniffer to read the data being transferred from the Wii to the WiiMote; they have been kind enough to make at least some of this data available on the internet through the WiiMote project [13]. Due to this being relatively new ground, a degree of reverse engineering and trial and error was required in order to make sense of the data once decoded from the WiiMote.

3.3. Interfacing in C++

There are three basic ways of communicating with the WiiMote programmatically. Once the device is paired over Bluetooth to the computer, it appears to Windows as a normal HID device, of a Joystick variation. This allows the use of any available HID library.

Microsoft provide HID communication libraries as part of the Windows Driver Development Kit which would have been the ideal solution as it gives full, low level control to the device. However, the HID library is not a part of the normal C/C++ windows development libraries, which are distributed with both Borland and Visual Studio, so there is not a Borland port and applications fail to compile. One way around this would have been to write a wrapper class in C using Visual Studio and export that to Borland.

On the other end of the scale there are third party WiiMote communications libraries which could be used. Several of which are available for free as both open and closed source [14]. But this would mean very little low level control over the device, not to mention hiding away the commands and control as well as in some cases, calibration of the data. This could potentially be useful as it can speed up development, but, many libraries prohibit commercial use and a strong understanding of the underlying communications is highly useful anyway.

The approach chosen was to use a different method to both of these. As part of Borland's development environment, there are objects called 'components' which can be used in a visual manner to build a graphical user interface to software, much the same as in Visual Studio. However, it also allows for the use of none-visual components which are represented by a visual icon in the design during development but which are hidden when an application is executed.

Many people and companies produce Borland compatible components. One of these projects is the JEDI Visual Component Library (JVCL) [15]. The JVCL project produces a package of many components designed to do many tasks – from building on basic included components such as a more advanced or intelligent text box to new components all together such as the one used to interface with the WiiMote.

Rather than just being a WiiMote library, the `TJvHidDeviceController`, provides a simple, Borland compatible, easy to use method of interfacing with HID devices. In order to use this component, it is simply dropped onto the design form of the application and event handlers are attached. This approach is a good balance between the low level access through the Windows DDK, which is quite complex and incompatible with Borland and an abstracted 3rd party library which doesn't give fine control over communication with the device.

The most relevant handler is `Enumerate` – when the component is told to enumerate, this function is called for every HID device attached to the system. The developer can then opt to open and connect to this device, attaching a device data handler or ignore it. There are also a variety of functions for dealing with devices being hot-plugged without being shutdown first.

The WiiMote is detected when the `HidController` is told to enumerate the attached devices. For each attached device, the enumerate handler is called with a pointer to the device in the form of `TJvHidDevice` passed as a parameter. This contains device identifiers such as vendor and product IDs, which for the WiiMote are:

Vendor ID: `0x057e`

Product ID: `0x0306`

Once the correct type of device is found, the developer can call the `Device->CheckOut()` method to mark the device as in use. That done, sending data to the device is a matter of using the `Device->WriteFile()` method and receiving data is via a data handler callback attached to `Device->OnData`.

In attempting to connect to the device through software, initially, all that could be received was status data from the buttons. All the attempts to set the report type to enable accelerometer or IR data were not working. After some formidable research, it appears that one of two things is happening – either there is a bug in the Microsoft Bluetooth stack which makes the `WriteFile()` function for HID devices fail, or the WiiMote is not 100% HID-Bluetooth compliant which makes it function intermittently.

It is likely that both of these are true, since there have been reports of a work-around using the Microsoft Bluetooth stack, but no details as to its implementation. In order to continue with work on this project, the Bluesoleil Bluetooth stack was used, this works very well with the WiiMote and is recommended within the WiiMote development community. Unfortunately, the Bluesoleil stack is not free, but is a requirement for this project to work properly and to enable two-way communication with the WiiMote.

3.4. WiiMote Messages

As the WiiMote is a (mostly) HID compliant device, it uses HID reports to determine the format of the data sent back to the computer and as a header on what type of data is being sent to the device. The WiiMote has several different known reports, only some of them need to be dealt with here – namely those involving the Infra Red camera and the accelerometer. As a by-product of this work, the buttons, LEDs and rumble motor can also be controlled.

There are two basic types of report – input and output. Output reports are those which come from the HID device itself. If the device allows these to be controlled in any way, it is via Input reports sent from the host computer. For example, the WiiMote's output report type is set by

sending a 3-byte message consisting of the following: `0x12`, `0x00`, `0xXX` where `XX` represents the report type which the WiiMote should switch to, in the case of enabling the accelerometer data, it is set to `0x31`. Although only there are only 3-bytes of data in the message, a full 22-byte message needs to be sent to the WiiMote (with the rest of the values set to NULL) in order for the message to conform to HID standards [12].

The output reports for the WiiMote always contain a 2-byte bitmask of the button status, this is normally necessary so that applications can take advantage of the button data while the accelerometer or IR camera is active. Examples of other input reports are `0x11` for controlling the LEDs on the bottom of the WiiMote – the data payload is simply on-off Boolean values contained within the last four bits of the payload byte.

There are too many different reports on the WiiMote to detail in full and those are just the ones that have been reverse engineered by academics and the open source community. Specific details for acquiring and decoding the data required are shown later.

3.5. Data Sources

There are three main sources of data from the WiiMote, these consist of:

- The buttons
- Accelerometer
- Infra Red camera

3.5.1. Buttons

There are 12 buttons on the WiiMote, the status of all of the buttons is transmitted as the first two bytes of every output report the device can send, in a bitmask format. The status of all the buttons apart from 'power' is obtainable via bit-mask decoding.

Byte	Bit 7	6	5	4	3	2	1	0
0	-	-	-	Plus	Up	Down	Right	Left
1	Home	-	-	Minus	A	B	One	Two

Table 2: Bit packing of WiiMote button status

See Appendix 9.5 for an explanation on bit-masking and the bit-wise AND operator.

3.5.2. Accelerometer

The Nintendo WiiMote features a 3-dimensional linear accelerometer [16] with a range of ± 3g. The data is acquired from the WiiMote by enabling one of several output reports which contain the accelerometer data. Any one of reports `0x31`, `0x33`, `0x35` or `0x37` contain the data, along with the 2-byte button status and any other data attached to it.

The accelerometer data comes in the form of 3-bytes, one for each axis, at bytes 2, 3 and 4 in X, Y and Z respectively. The output report rate from the WiiMote can be adjusted, up to 100Hz, allowing for highly accurate measurements to be taken. Unfortunately, little is known about how the unit is calibrated, however, it is possible to derive usable values as when the WiiMote is placed on a flat surface, the upward facing dimension of the accelerometer will be represented by 1g, the effect of gravity.

In order to calibrate the WiiMote, first, the zero points must be calibrated, to do this, the raw values in X, Y and Z must be taken when the WiiMote is at rest facing in each of the dimensions. Figure 3 shows the axis orientation with respect to the WiiMote. The raw values are then taken when the WiiMote is:

- Horizontal – the buttons on the front are facing up and the WiiMote is resting on a flat surface
- Vertical – the infra red camera on the top is facing down and the WiiMote is stood on a flat surface
- Side – The left side is facing up with the infra red camera facing away and the buttons facing to the right, with the right side on a flat surface.

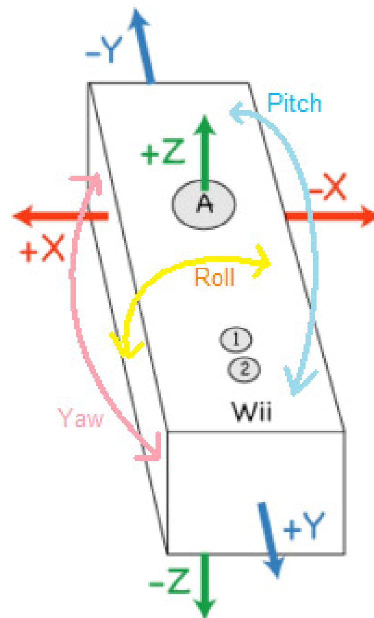


Figure 3: WiiMote axis reference [16]

This gives 3 values for the raw acceleration in each direction. This means that for each axis, 0g is represented twice. This is evident from the typical values:

Position | Value	X	Y	Z
Horizontal	130	131	152
Vertical	130	157	127
Side	155	130	127

Table 3: Typical raw values for acceleration in each dimension

Table 3 clearly shows when the 1g gravitational force is acting in each dimension, i.e. when the WiiMote is placed flat on a table, only the Z-Axis shows force acting on it, where the downward force is caused by the effect of gravity. From visually inspecting the data, it is evident that 0g is represented by a raw value of around 130, and 1g is around 154.

In order to mitigate error, the value for 0g on each axis is averaged. These three values are used for calibrating the output value in terms of g. Normally these would be written back to the WiiMote's internal memory, however, since little is known about how this functions, it is corrected for in software instead. The nine raw values are stored locally to the computer running the application.

In order to calibrate the force in terms of g, the following equations are used,

$$x = \frac{(x_{raw} - x_0)}{(x_3 - x_0)}$$

Equation 1

$$y = \frac{(y_{raw} - y_0)}{(y_2 - y_0)}$$

Equation 2

$$z = \frac{(z_{raw} - z_0)}{(z_1 - z_0)}$$

Equation 3

Equations 1 through 3 calculate the final acceleration value in g. The calculation is made using floating point numbers to allow greater accuracy. Testing the data empirically, with the calibration in place, seems to confirm the claimed specification. There is of course an error as the values should be written into the WiiMote which would place the zero points all at the same raw value, however, for the purposes of use within this project, this calibrated value is more than sufficient. It shows acceleration, direction (i.e. positive or negative) and a degree of magnitude in each of the three dimensions.

3.5.3. IR Sensor

The second main data source within the WiiMote is the Infra Red camera, commonly mistaken as an infra red emitter, to use with the ‘sensor bar’, the camera is actually a high resolution camera manufactured by PixArt. It features a resolution of 1024x768 with up to a 100Hz refresh rate (the same as the accelerometer) and a 45º horizontal viewing angle. It also has a four object multi object tracking engine [16]. The camera is sensitive to only infrared light, but different variations on infra red wavelength have been experimented with, with varying degrees of success [18].

It is worth noting here that the next nearest available cameras with this sort of specification very quickly start costing over £100. Even just on the resolution this is remarkable, but with this sort of refresh rate make the WiiMote very useable in many areas.

Enabling the IR camera is significantly more complex than the accelerometer. It occurs in four stages,

- Enable medium-length IR data reports (`0x33`)

- Enable IR clock
- Enable IR camera
- Write configuration data to internal memory

Little is known about the configuration data for the infra red camera, just that it controls parameters such as gain and detection parameters for infra red object size. Optimal values are unknown, however, the open source community have empirically tested and derived values which are likely to gives good results [18].

Enabling the IR clock and camera are simply a matter of sending the correct input reports with the correct parameters. The output data format can be presented in 3 formats, short, medium and long. The medium format is most usable here in the form of report type `0x33`. This report contains the IR data as well as the 2-byte button status data and the 3-byte accelerometer data. The full report is structured as follows,

Byte	Bit 7	6	5	4	3	2	1	0
0	-	-	-	Plus	Up	Down	Right	Left
1	Home	-	-	Minus	A	B	One	Two
2	Accelerometer X							
3	Accelerometer Y							
4	Accelerometer Z							
5	IR0-X							
6	IR0-Y							
7	IR0-Y MSB		IR0-X MSB		Size			
…								
16	IR3-Y MSB		IR3-X MSB		Size			

Bytes 8-16 follow the same format as 5-7

Table 4: Report 0x33 Structure

As is evident from the report structure, the X and Y position values do not fit within an 8-bit byte and instead are split over multiple bytes. Structurally, these are 10-bit values. These are combined together through a combination of bit shifting and the `OR` operator. Initially it was thought there was no MSB (most significant bit) value, and the position value simply 'wrapped' at the maximum value of 255. See Appendix 9.2 for an explanation on the OR operator and Bit Shifting.

Although the output values for both X and Y positions are 10-bit, experimental testing suggested that the full range was not used for the Y co-ordinate, with it going out of range at around a maximum value of 760. When an IR source goes out of range of the tracker, the output values go to maximum. Research into the physical hardware of the device confirmed its maximum resolution, so when scaling into a percentage or fractional value, the correct divisor for the Y axis is 768, with the X axis remaining at 1024.

Being able to track objects in 2D is all well and good, but sometimes 3D tracking is required, to do this normally requires stereoscopic vision – which is how human depth perception works, using two eyes. In Wii systems, 3D positioning is acquired through the use of the 'sensor bar'. The name 'sensor bar' is a misnomer as it doesn't actually sense anything; it is a device which contains two clusters of infra red LEDs at a fixed distance apart. The official Wii devices are powered from the Wii, unofficial third party accessories are battery powered. It is also possible to build one, using simple infra red emitting LEDs. The fixed distance between the LED clusters is known by the software so that it can determine the distance based on how close the objects appear to be.

3.6. Use of the data

Within the application, there is only one use for each type of data available. The IR camera data is used for positioning objects on screen and the accelerometer is used to derive the velocity value for a MIDI note.

3.6.1. IR for On-Screen Positioning

Using the infra red camera to position objects on screen is very simple, it is just a matter of taking the raw X and Y positional co-ordinates and scaling them to screen values. Borland provides a simple way of retrieving the current screen resolution by accessing `Screen->Width` and `Screen->Height` from within the application code.

To set the on-screen position of a Borland visual objects, they have `Object->Top` and `Object->Left` properties which are linked to their positions on screen. Initially, the IR data was mapped to progress bars to give an indication of the values of the incoming data, before being mapped to a small window marked 'IRz' where 'z' was a number representing the IR object being tracked. The data was ultimately used for setting the on screen position of a drum stick image as well as being fed into the triggering algorithm. The algorithm determines if a drum 'hit' occurred or not.

3.6.2. Accelerometer for Velocity

The accelerometer within the WiiMote has many applications. Here the downward acceleration force (on the WiiMote's Z axis) will be mapped to MIDI velocity. In order to make the velocity feel more natural, the calibrated accelerometer value is multiplied by a sensitivity value to restrict the range of velocities which are used.

Although at this stage of development, the sensitivity value is fixed, there is scope to make this user adjustable, much like the controls on many touch sensitive keyboards. The MIDI velocity is derived when a note is triggered from the IR trigger algorithm.

3.7. Algorithms

Only one main algorithm is really used to provide user interaction – that is the Infra Red trigger algorithm which is based on the users' movement of the WiiMote to control an on-screen virtual drum stick. There is an additional algorithm which does not have any involvement in the users' interaction but is included out of research interest.

3.7.1. IR Trigger Algorithm

Throughout the execution of the application, the user can control the on-screen movement of two virtual drumsticks. The algorithm uses quite a simple threshold triggering based method whereby the previous and current positions of the object are monitored. In the case where the previous value is less than the threshold and the current value is greater than the threshold then the trigger is activated.

The triggering algorithm runs within the WiiMote C++ interface class and so operates entirely on floating point fractions and is independent of the screen resolution. An on-screen visual representation of the threshold is used to assist the user in stopping movement of the WiiMote just as they cross the trigger line – this generates the greatest force within the accelerometer as the user has stopped suddenly, i.e., producing a greater MIDI velocity. The velocity is calculated as detailed above using the accelerometer value which is also passed into the function. In order to allow for different types of trigger, the X-axis IR position is used to determine which quarter of the horizontal screen the trigger occurred in. This determines which of four drum sounds is used, either, hi-hat, snare, tom or crash cymbal. Only the tracking of IR object number 0 is used in the algorithm, but two WiiMotes are used to provide two drum 'sticks'. An alternative way of approaching this would have been to use a single WiiMote, but use two IR light sources, e.g. IR pens which the user moved about, although this would not have allowed for the use of the accelerometer to determine velocity.

Figure 4 represents the triggering algorithm in operation,

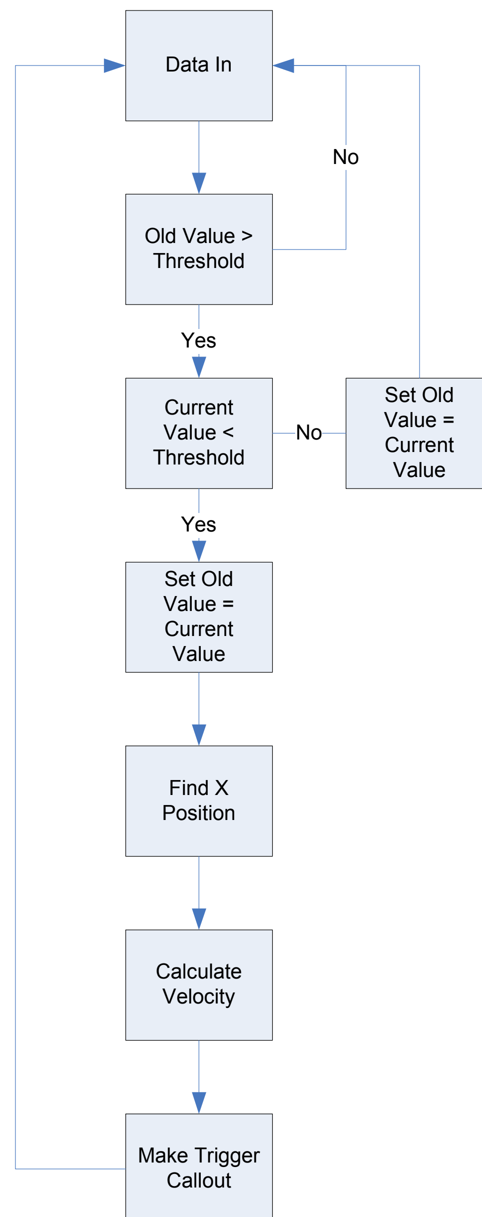


Figure 4: Trigger algorithm operation

The triggering algorithm is called every time new IR report data is received by the application from the WiiMote. It only takes the floating point values for the location of IR object 0 and Z-Axis acceleration. Once any decisions within the algorithm have been made, it returns and waits for the function to be called again.

3.7.2. IR Direction Sensing Algorithm

The other main algorithm which has been designed and implemented into this system is a basic direction sensing algorithm. Although the raw numbers of the IR locations can show that the object is moving and even be mapped to screen co-ordinates, it does not give its direction as a discrete value – i.e. is it going up, down, left, right or stood still. The main problem with this is that the data is never stationary – it is always moving by a small amount, therefore at a minimum, the data must be smoothed to allow calculation of direction.

The easiest way of achieving this is to calculate the velocity on each axis. Since velocity will be positive or negative depending on direction, it is a simple calculation from the position, or distance which is already available. The direction finder algorithm keeps a ring buffer of the last 50 position samples in each axis before averaging the first 25 samples and then the last 25 samples. From these two average values, the gradient can be calculated, since this is essentially the derivative of distance with respect to time, this gives a velocity value.

The velocity value is then checked against pre-defined thresholds to determine which direction, if any, the object is moving. If the incoming data was for the X axis, it would be possible to determine horizontal movement left, right or stationary, for the Y axis, the outcome would be up, down or stationary.

The two output values could be combined to give a single directional movement which has 9 directions, similar to the D-pad style controls used in computer games. Figure 5 shows the operation of the direction sensing algorithm in flow chart form.

At present, this data is not used, but the output is displayed as part of the WiiMote configuration and debug screen. The number of samples used within the ring buffer determines how quickly the algorithm can detect a change in direction, after some testing, the value of around 50 was shown to give results which responded quickly to direction change but which were still smooth enough to give a consistent reading. Since the WiiMote can respond with IR data at up to 100 times per second, the amount of user time it takes for the reading to change is a mere 500ms.

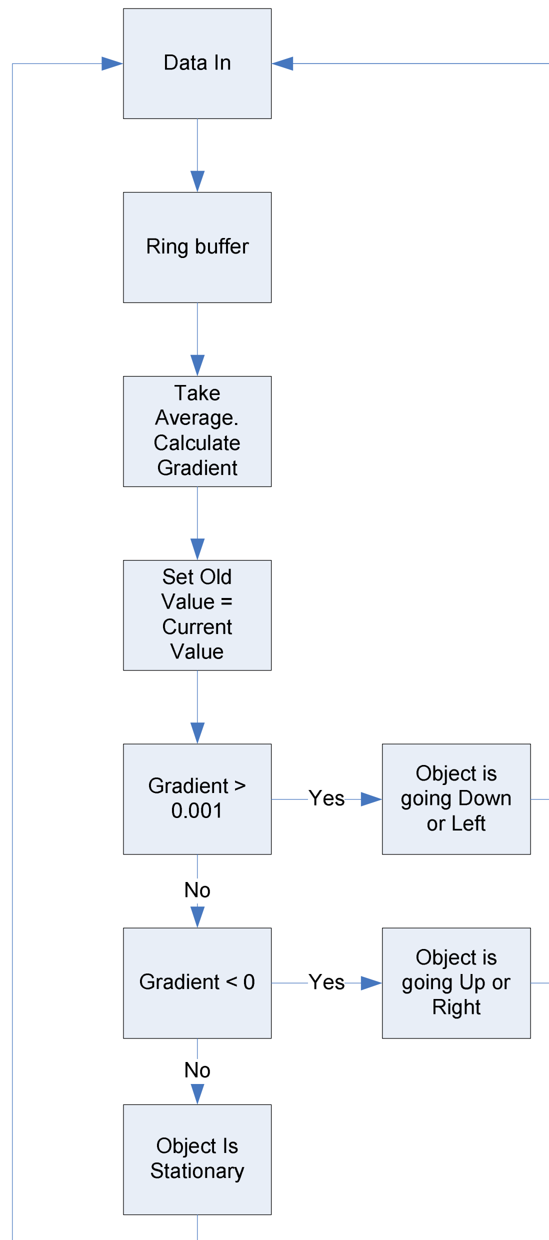


Figure 5: Direction sensing algorithm operation

CHAPTER 4. SYSTEM DESIGN

4. SOFTWARE DESIGN

In the modern world, software is everywhere, as such it is more important now than ever to not just use modern software design methods and techniques, but to embrace them and take them to their full potential. Some of these techniques have been employed in the design and implementation of the WiiMote MIDI-Drums system which became the target of this project. This chapter will discuss the design of the system and why such decisions were made along with some details of how the product developed through the various stages of prototyping and implementation. The final code for the main and associated applications is contained within Appendix **Error! Reference source not found.**.

4.1. Prototyping to Final

In any piece of software design, it is important to understand the concepts of what is trying to be achieved, that is to say, it is no good to design software without first understanding how two different things are to interface with one another – for example, how to interface with a MIDI device, what data needs to be transferred and when, as well as what is required to initialise the system before use.

Chapters 2 and 3 have dealt with the initial prototyping of functionality, interfacing first with the MIDI hardware and then with the WiiMote. A widely used approach to prototyping interfaces is the ‘everything in Unit1’ approach. ‘Unit1’ refers to the Borland default name for the first source file in a new project. So when writing new functions or interfaces they almost always appear in this file, usually with cryptic names. This is exactly the model used to develop the software in this project.

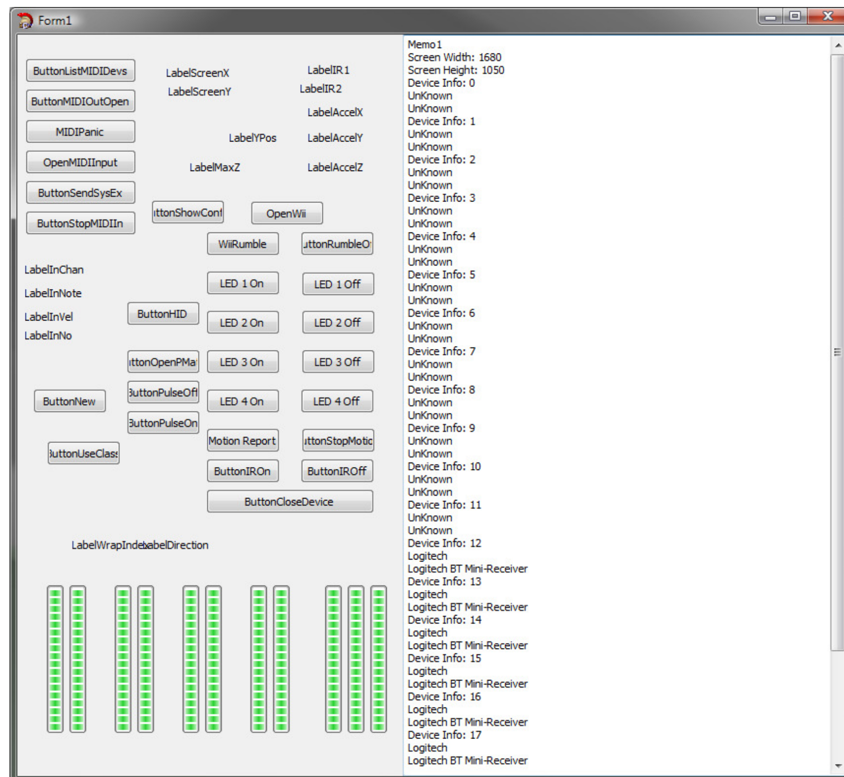


Figure 6, Development application

Figure 6, shows revision 47 of the application where many of the functions are directly mapped to buttons with a large debug log window. As is evident, there is a button for each of the WiiMote functions such as LEDs, rumble motor and so on as well as MIDI functions. The debug log window will display information about attached devices as well as any errors which occur. The buttons and function names often have cryptic names during the prototype stage as usually only a single developer needs to understand its function. When development is ready to move into a broader structural stage, the code is split out into modules or classes to make them easier to deal with and be ready to be integrated into other peoples' code.

4.2. Software Structure

Now that classes have been created to interface with MIDI and the WiiMote, they were fitted together to form a single piece of software along with a user interface. Figure 7 shows the full overall application structure.

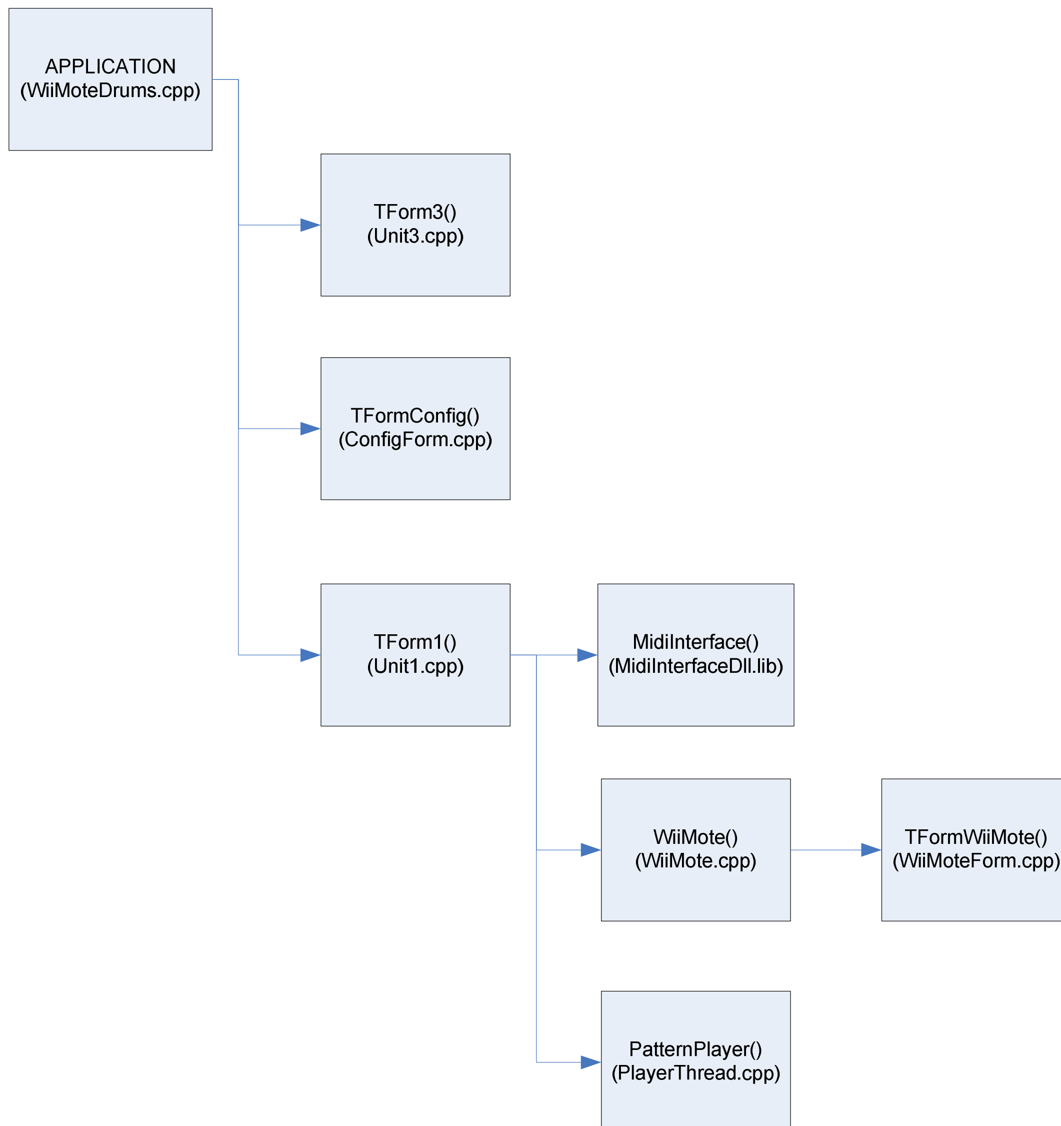


Figure 7: Application structure

Each box in Figure 7 represents a different C++ class object which has been written as part of this project, Borland components and VCL classes are not included. The arrows represent the creation order of classes and who is the owner of the data. All applications have a 'main' function. In this case it is taken care of automatically by Borland in the form of WiiMoteDrums.cpp which just contains basic functions to start up the rest of the application and in the correct order.

TFormConfig is a simple user-configuration form for choosing the ID numbers of WiiMotes used to represent left and right drum 'sticks' as well as the MIDI output hardware to be using.

Attached to each of these controls is a Borland callback, or function pointer. See Appendix 9.6 for an explanation on function pointers.

TForm1 is where most of the application runs and it is this class which creates an instance of the MIDI interface plus a WiiMote class for any attached WiiMotes. These are enumerated using the `TJvHidDeviceController`. The WiiMote class also produces a visual window, mostly for debug, but also to use as control over the incoming data such as enabling the IR camera and accelerometer reports.

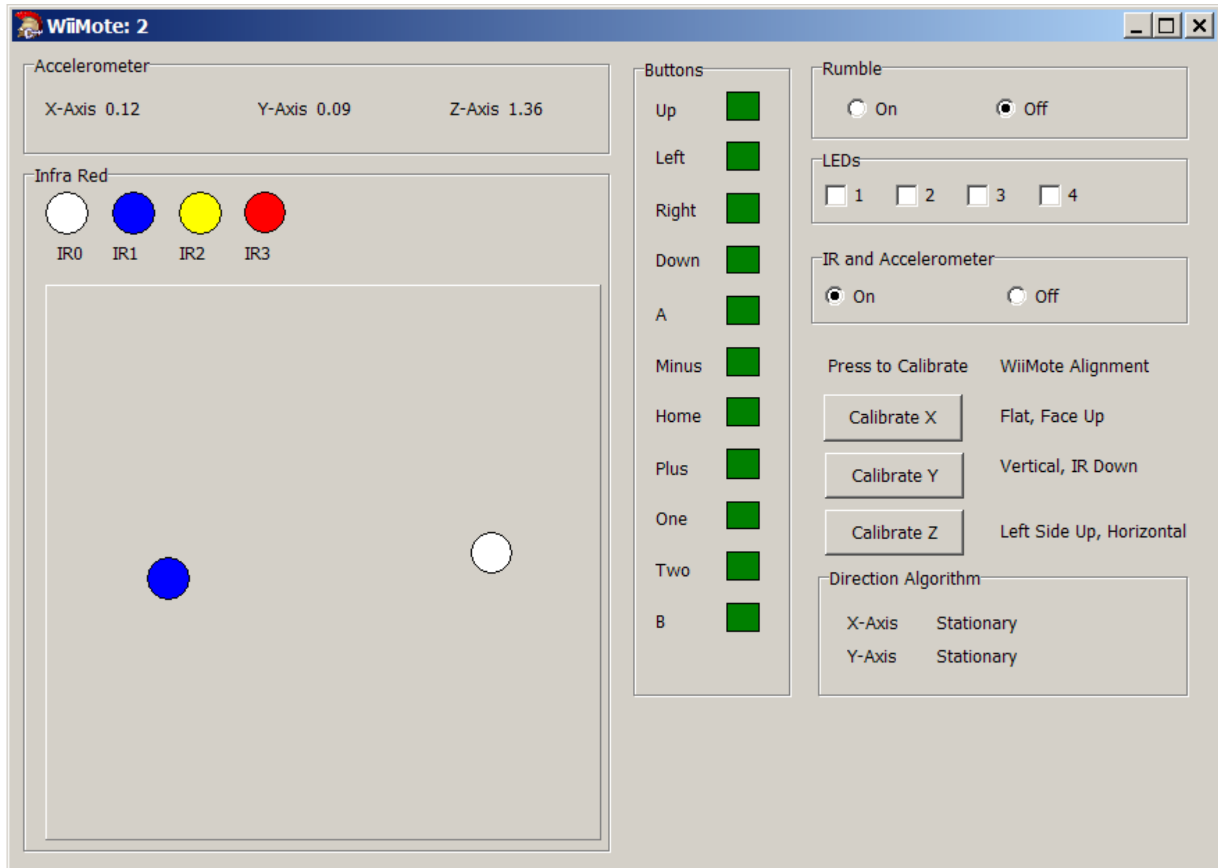


Figure 8: WiiMote window

Figure 8 shows the window which is created for each WiiMote, here the data is shown in neat, calibrated form with coloured spots to represent the IR objects which the camera is currently tracking (the minimum and maximum values are scaled to the bounds of the box). Also provided are buttons to calibrate the accelerometer within the WiiMote. These values are written to disk in an .ini file so that they are automatically loaded in the future.

Unfortunately, the only way to identify the WiiMote is through its device enumeration ID, which can change between systems and if devices are added or removed from systems. Ideally, the

WiiMotes would be tracked through their serial number or MAC address, but accessing this portion of memory using the BlueSoleil Bluetooth stack causes a Windows blue screen driver memory error and so, like in much production software, this ‘bug’ has to be coded around and an alternative method was developed.

The final new class written for this project is PatternPlayer. This is concerned only with the user testing of this application and is not intended to be used in part of production code. It is simply a mechanism for playing back drum patterns and does not use an optimised method for doing this, hence why it has been placed into its own thread, to mitigate timing issues, which unfortunately, were still prevalent. The detail of the pattern playback thread will be covered in the next chapter with the user testing.

4.3. System Lifecycle

An important part of software design is being able to understand the software life cycle. The software lifecycle is a way of describing what needs to happen and when, in normal running operation. Figure 9 shows the life cycle of the demo system created to demonstrate and test the WiiMote and MIDI interfacing classes.

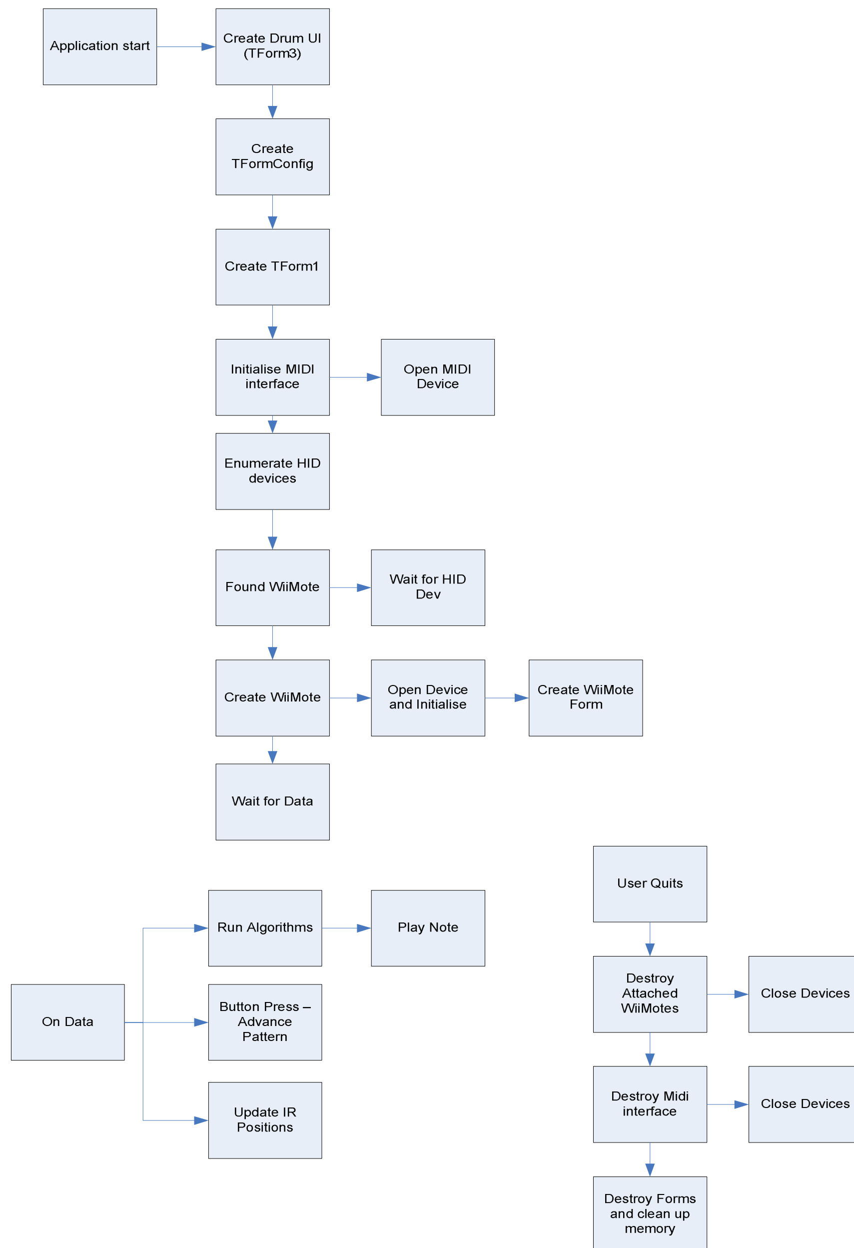


Figure 9: Software lifecycle

Due to the complexity of the system, it is not possible to show the full detail of the software as it operates, the diagram does however, cover the general operation. The previous chapters cover the MIDI and WiiMote classes, with their algorithms in more detail.

There are three distinct stages to the life cycle of a system, commonly referred to as set up, operation, and tear down. As implied by the name, set up refers to all of the actions which much occur before the system is ready to accept incoming data – whether that be data from a device

(the WiiMote), data from the user (configuration) or data from another class trying to access something (the callbacks). In interfacing with the WiiMote and MIDI there is quite a lot of setup which needs to be carried out in order for it to work. The initialisation of the WiiMote includes attaching callbacks for incoming data as well as sending commands for enabling the IR camera and accelerometer.

Once the application is in running mode, it is more 'reactive' in its action. Only when new data arrives into the application does it process anything or run the algorithms. In this case data is being streamed into it from the WiiMote so the application is always busy, this is not always the case and some applications will just sit and wait for user input.

Once the user decides to exit the application, pressing the quit button closes down all of the interfaces, destroys created objects and generally cleans up the memory. Created `WiiMote` classes are stored in a vector so that at shutdown the application destructor can run through the list and delete them one by one. This vector also serves as a way of checking if a particular WiiMote has already been created, should the HID devices be re-enumerated.

4.4. Modular Design

The aim of the design of this application was modularity, the two main classes in use – `MidiInterface` and `WiiMote` are both completely independent classes and could easily be slotted into another application. A move on from this would be to have a `WiiMoteManager` style class which wraps the `TJvHidDeviceController` processes so that the developer does not have to also create and install that component. The WiiMote class should also have at least the ability to disable the debug/data window for use in command line and windowless applications.

CHAPTER 5. TEST SYSTEMS

5. SOFTWARE TESTING

In commercial development, application testing, both technical and user is highly important. Testing allows developers to make an assessment of the usability of a product and its technical stability. The application which is the focus of this project had both type of tests applied to it.

5.1. Technical Testing

Technical testing is largely an automated technique employed by developers as a means to test their code without their interaction. Since large projects generally have automated software builds, they can easily add in automated testing to execute alongside. Testing covers basic input and output parameters plus memory leaks and error handling.

For example, a function which returns true for an input of an odd number and false for the input of an even number,

```
bool __fastcall Math::IsOdd(int aNumber)
```

Then the written test would check the result is what is expected for given input data. In Borland, testing is assisted by the use of the unit testing framework, DUnit. Unit testing is most useful in object oriented code, here it was applied to the class of `MidiInterface` and provides functions for checking results in the form of `CheckTrue()` and `CheckFalse()`. Either of these two functions failing shows up as an error in the auto generated user interface. There is also the option to build the test as a command line system with the results logged to disk for more automated systems.

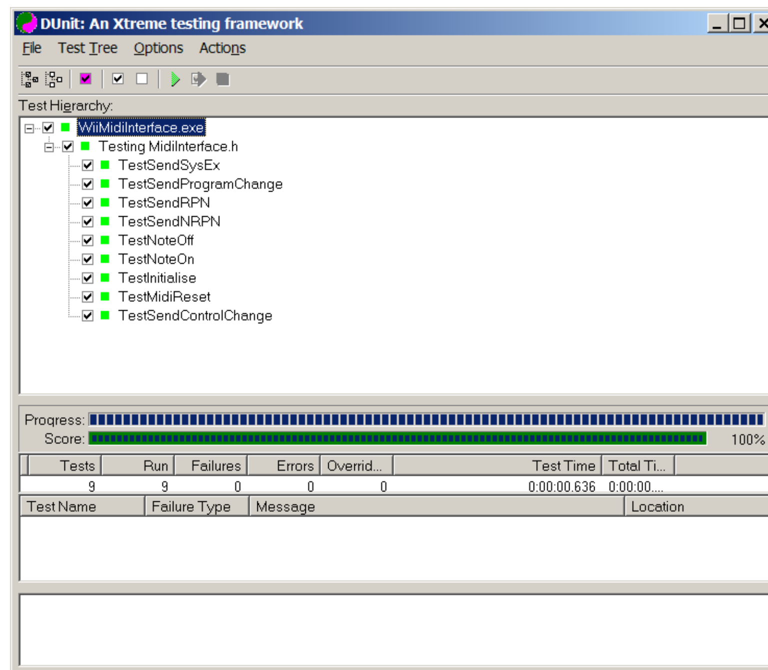


Figure 10: DUnit test for MidiInterface.

Using the example test from above, the full test as written in the DUnit framework would be the following,

```
void __fastcall TTestMath::TestIsOdd()
{
        CheckTrue(Math->IsOdd(1));
        CheckFalse(Math->IsOdd(2));
}
```

The test function is accessing a class called ‘`Math`’, the hypothetical object being tested. In order to be able to access it, there is a ‘setup’ area within the testing framework where anything which needs to be carried out in order for the tests to normally succeed can be placed. In this case it simply requires a new instance of the `Math` class to be created.

As well as the expected result from known input, it is common to also try inputting erroneous data to see if the function handles it appropriately, crashes or gives other unexpected results. If an application is likely to produce exception errors (either by intention or accidentally) it is possible to wrap the function calls into `try…catch` statements so that exceptions are caught and reported as errors instead of taking down the entire test framework.

Once tests are complete, there is the opportunity to tear down the session and delete any objects which had been created before the object could be tested, in this hypothetical case that would be the `Math` object.

When running the tests for the MIDI interface, each test consisted of using ‘good’ data for each function parameter then in turn passing bad data in each parameter – generally this is simply an out of bounds error where the input number is greater than what is valid, e.g. a number greater than 16 for MIDI channel is invalid.

When testing this class, it became apparent that the boundary checking on the function parameters was faulty and did not work properly with the functions returning OK values when clearly the input data was erroneous. Without testing in this manner it would have been difficult to detect this, now the return values indicate an error when the input data is wrong, this is useful because it provides a means of checking to see where an error is occurring if an application is nor performing as intended.

As well as function input and output tests, Borland also provides a means for testing memory leaks. Memory leaks are where system memory has been allocated, in the form of a data block or object which has not been deleted before the application has terminated. Creating a memory leak is easy, simply executing

```
unsigned char* Data = new unsigned char[100];
```

Will leak a block of 100 bytes, unless it is matched, somewhere in the code by

```
delete[] Data;
```

The delete code, executed typically when the data is no longer of use, or at the end of an application, cleans up the memory and marks back to the operating system that it is free. When enabled, the Borland monitor, Codeguard, will monitor all memory allocations and accesses so that when the application is ended it can produce a report containing information on any memory which was not cleaned up. It also checks to make sure that the application does not access invalid memory such as pointers or objects which have previously been deleted but the code is not checking to see if that object is still valid before access.

CodeGuard is a valuable tool for making sure memory leaks do not occur, they are important because if a system is intended to be running continuously yet it is leaking memory, very quickly the host computer will run out of application memory and will typically crash. In real-time and critical systems such as broadcast audio video, military, and medical applications, reliability is usually much more important than applications’ ease of use.

5.2. User Testing

With any software system which is to be operated by humans, the user interface plays a large role in the usability of the software, some would say the greatest. To this end, once the project had reached a suitable point, real world usability testing was required.

5.2.1. Software

In order to have some way of determining usability in a more scientific manner, some regulated repeatable tests were required. In the case of the WiiMote Drums, the most suitable way of achieving this is to provide a 'play along' style system. The application plays back a series of drum patterns which have been pre-programmed and the user is to learn to play along to the pattern while the application records its results.

This is similar to games like Guitar Hero where the user has to press coloured buttons in response to the visual stimuli. In this system however, the user has to use the audio stimulus and physically move the WiiMote to 'hit' the correct 'drum' on screen. The user is provided with a visual representation of the virtual drum stick location in order to provide some visual feedback as to what they are doing.

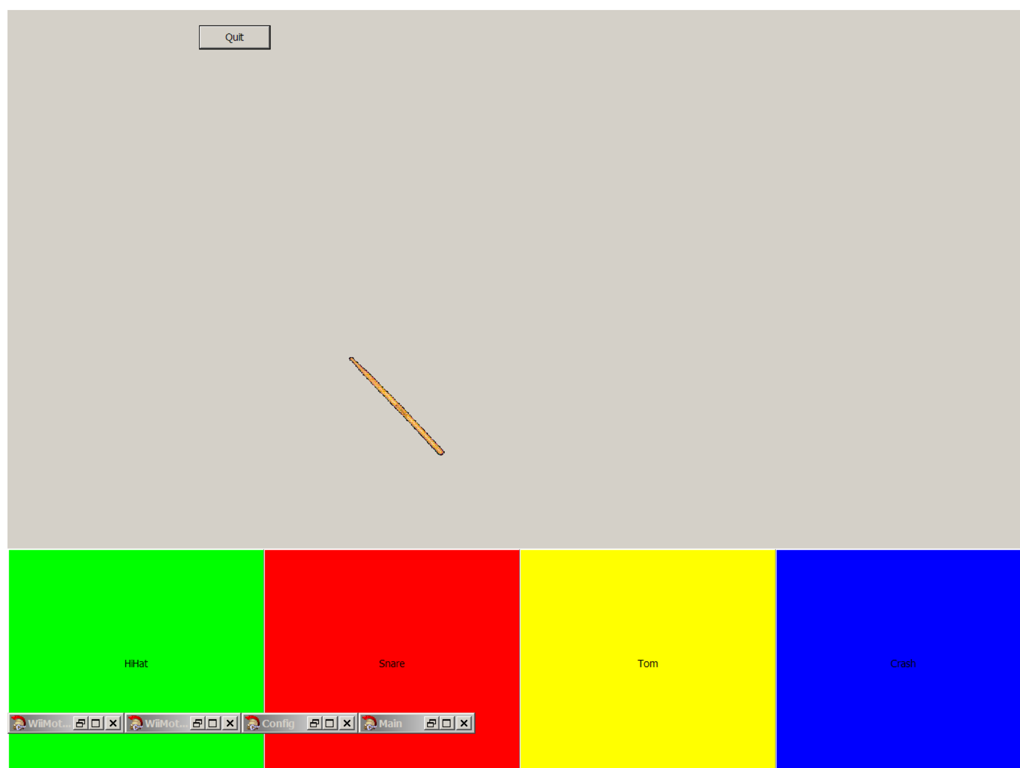


Figure 11: Demo application main screen

Figure 11 shows the main screen of the system. Four coloured ‘pads’ or ‘drums’ are along the bottom of the screen to represent Hi-Hat, Snare, Tom and Crash Cymbal.

The drum patterns required three things:

- MIDI interface – taken care of using the MIDI interface library.
- Playback mechanism or scheduling of notes
- Storage of patterns

The pattern storage is on the face it of, the simple part of the system, however, in order to allow for easy design of playback functions, some custom storage structures are required. The software will create an array of ‘Patterns’. A ‘Pattern’ is a structure containing an array of maximum 100 ‘Notes’ and an integer value representing the number of notes.

```
typedef struct
{
 int Note;
 int Time;
} Note;

typedef struct
{
 Note Notes[100];
 int NumNotes;
} Pattern;
```

This way an individual note can be accessed via `Patterns[PatternIndex].Notes[NoteIndex].Note`. Each note has its MIDI note number and a time value. The time value is measured in milliseconds from the previous note, the playback scheduler uses this to determine when the next note should be played.

Initially, the application made use of the OnIdle event within Borland. The OnIdle event is a function which is executed whenever the application is not busy in any other function or dealing with user input. Here, a Windows Waitable timer is set [19] and when the handle to the timer is signalled, the next note within the pattern is sent out of the MIDI device using the MIDI interface library.

A Windows Waitable timer is a device provided as part of the standard Windows libraries which allows applications to set timers based on the built in Windows kernel clock. The function allows setting of both absolute time and relative time. Whenever the application is in the OnIdle loop, it checks the handle to the timer to see if it has been signalled.

Unfortunately, this is not a very good way of tackling the issue of note scheduling for playback and as such it suffered a great deal from mistiming when other events were occurring, such as having to re-draw the on-screen drum sticks. To mitigate this issue, all of the playback code was moved into its own software Thread. A multithreaded application is more useful as the users CPU can run both tasks side by side, this means that events occurring in the main part of the application should not impact on the scheduling of the playback thread. Using a thread mostly solved the problem, but to provide proper playback within the application a different approach should be used.

The thread does not contain any thread safety measures to ensure that two threads do not attempt to write to the same memory simultaneously, fortunately, this is less of an issue in this case as the only data which needs to pass into the playback thread is a message to move to the next pattern within the set. Since this pattern playback mechanism is provided purely as means to carry out the subjective tests, no further work is really required and the class and functionality should be removed.

5.2.2. The Tests

The tests themselves consisted of 10 different drum patterns, with increasing difficulty designed to test the ability of the users to play back both in time and the correct sound. The basic pattern begins with simple equal timed snare hits building up to multiple notes and simultaneous hits on different drums.

The users were given a few minutes to play and practice with the system before beginning the tests. There were no time limits in place so that the user could listen to the pattern and then follow along, repeating as much as they liked. Once the user had decided that they had either completed a pattern or given up, they pressed the ‘B’ button on the back of the WiiMote which was linked to increase the current pattern playback index within the software so that the next pattern began.

The users were given an explanation of what was to happen before completing a short questionnaire and carrying out the tests. After the practical tests, a few final questions were asked. A sample questionnaire is included in Appendix 9.7. As the users attempted the tests, the software logged every note hit by the user and every note by the pattern playback system along with absolute time, velocity, pattern index and user index. These log files were used as the basis for the analysis of the results. A sample entry from the log file is shown below,

```
Computer,3,2,38,100,3948572
```

The fields are as follows

```
Type, SubjectID, PatternID, Note, Velocity, Time
```

The comma separated value files can then be analysed using a variety of software including popular spreadsheet packages.

5.2.3. Analysing the Results

Analysing the data can be tackled in different ways, one approach would be to load the CSV log files into a spreadsheet package and analyse the data entirely there. This time however, due to the complexity of the analysis, approaching the problem from a software coding point proved to be easier. To this end, a simple application was written to analyse the results, it loads in both log files – one contains notes produced by the playback system the other is from the user input. It then allocates memory for each note ‘record’ and runs it through the analysis algorithm which outputs another log file. For each of the drum patterns, the log contains the number of notes in the pattern, the number on target, the number incorrect and the total number of notes. It will also calculate the percentage correct.

An example results table is shown below,

Test Subject 4					
Pattern	Notes In Pattern	On Target	Wrong Notes	Missed Notes	Percent Correct
0	28	18	137	10	64.29
1	50	25	1	25	50.00
2	32	21	0	11	65.63
3	32	12	1	20	37.50
4	39	13	0	26	33.33
5	120	37	13	83	30.83
6	7	0	0	7	0.00
7	238	56	14	182	23.53
8	97	55	3	49	56.70
9	230	72	6	160	31.30
Standard Deviation of Velocity			22.102827		

Table 5: Example results table

For a note to be classed as on target, the user must hit the same note as the computer within ± 100ms. The application uses a series of counters to maintain the numbers for each parameter and for each drum pattern. Each of the note 'records' also contains a Boolean 'Processed' tag so that notes are not marked more than once, this reduces errors when multiple notes are very close together of different types. The standard deviation of the velocities is also calculated as a means to determine how consistent the user was in triggering the drums. A full set of results tables is available in Appendix 9.8.

The log files produced by the analysis program can be imported into any spreadsheet package and it is then possible to plot a graph of the data for each user and each pattern simultaneously.

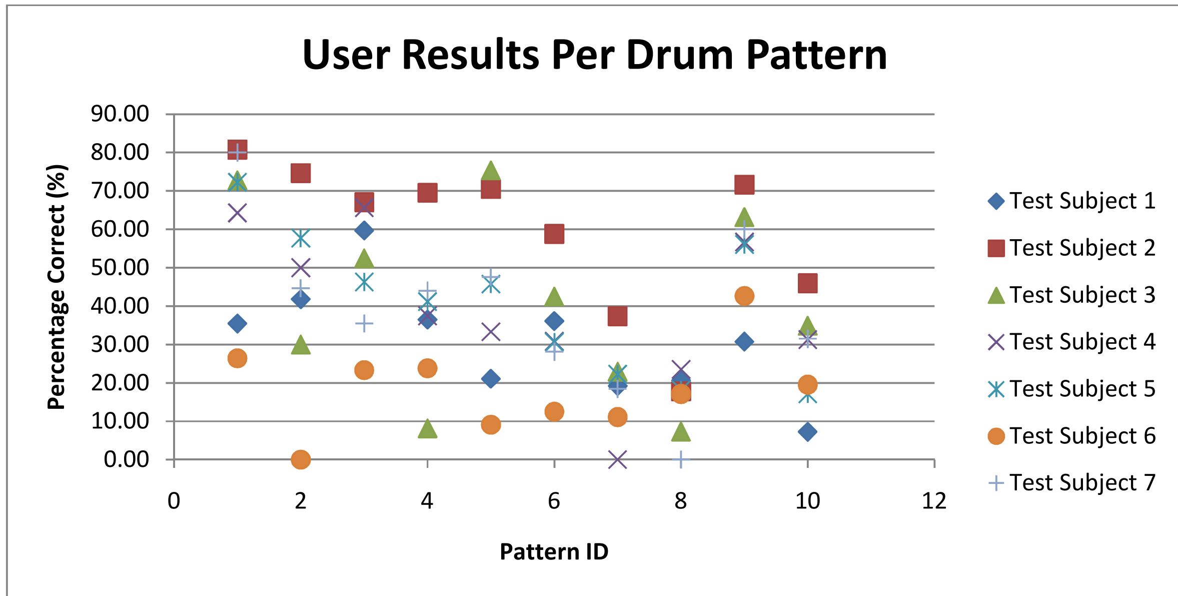


Figure 12: Results graph of all users per pattern

From the graph it easy to see that overall the results do not mean very much, however, there is a definite downwards trend as the pattern index increases, this will be caused by the increased pattern complexity. Also, test subject number two seems to be significantly more able than most of the rest of the subjects. As it happens, test subject number two is a trained musician and therefore has quite a good sense of rhythm and listening both of which are important to be able to do well in this test. This user also took their time to complete the tests instead of attempting to race through them. In these tests taking your time to listen and make sure the pattern is correct is more beneficial than attempting to hit the notes at random.

5.2.4. Results Usability

The results are quite subjective and do not show a lot as to the usability of the system, however, some useful ideas were created as a result of the questionnaire. Many of the users saw the potential to use this technology in computer games, or other gesture based interfaces – such as remote control in presentations or audio and video control.

Some of the problems highlighted by this testing is that there were no visual cues for following the drum patterns, however, while this may have been useful to aid in the users' tests, the system is not intended as a game as such, more of a way to be able to play a fairly realistic drum kit quietly and with little expense. One of the most useful suggestions was to curve the drums around the screen so their positions in space feel more natural.

CHAPTER 6. CONCLUSIONS & FUTURE WORK

6. CONCLUSIONS

The aim of this project was to produce a software interface library for MIDI and to find an alternative way of generating MIDI data from user input. Although both of these were successfully achieved through the MIDI interface library and WiiMote Drums there is still potential for further work in this field. No other system yet uses location positioning to simulate musical instruments using the WiiMote; they all require the user to press buttons to control the pitch element of the experience.

The motion direction sensing algorithm for the WiiMote developed within this project has scope to be developed into a gesture tracking system, with the addition of a second WiiMote and using them both in fixed positions (while moving the IR source) would allow reasonably high resolution tracking in 3D space. The 3D space could be mapped to MIDI control of pan in a surround sound mixer or within Grey's 'Timbre Space' [20] used in synthesis. In less broad parameters, even something as simple as a Theremin emulator could be created using this technique.

This project has produced the building blocks for future research in this field, which, if released to the community, could be improved and built on as part of bigger projects. The MIDI interface and WiiMote classes simplify the work required to add in support for either into an application. The MIDI interface class brings accessing MIDI hardware up to date using modern programming methods while the WiiMote adds in hardware support for a whole new class of device allowing a developer to access both processed data and triggers, or direct raw, but still calibrated, information from the hardware.

Although the project plan presented multiple options for which kind of hardware device should be implemented, only the WiiMote was developed upon. This situation arose due to the large interest which the WiiMote has received from open source software, electronics and musical communities. Much of the new research has only been published within the last two years so that when this project began there was significantly less information available than there is now, making it an interesting and exciting project to work on.

One of the alternative proposed input methods was to use the OCZ Neural Impulse Actuator, which would also have been a highly interesting project, however, very little information on use of the hardware is available and due to the type of data which would have been available from the device, it would have not been a viable project to work on within the time scale.

That said, the project still produced some excellent results using up to date and modern programming techniques as well as industry accepted practices. The technical and subjective testing, although on the face of it did not produce entirely usable results, were still useful in a more subjective manner in the ability to get a 'feel' for the system and how it would cope in the real world.

6.1. Further Work

Developing this project further can be approached from different angles, at a more direct approach such as by further developing WiiMote interaction, or more broadly by finding alternative sources of input data.

With the WiiMote there is work which can be carried out which can make use of the direction sensing algorithm and target tracking in 3D space. 3D tracking has already been achieved for following the motion of an orchestra conductor [4]. This kind of work can then be applied into either musical instrument simulation, similar to the WiiMusic game but where the full motion of the user is tracked in 3D so that the simulation is more real or into more general applications like head tracking for virtual reality rendering [16].

As well as traditional applications, such as musical instrument emulation, thinking more abstractly would allow people without musical ability to create or at least modify music. An example of this would be using the WiiMote to control pitch, tempo or volume depending on its orientation (using the accelerometer for measurement), anybody would then be able to move the WiiMote about and get an instant result – controlling for example, playback of pre-set loops. Work similar to this has been used as a means of browsing sound effects libraries [21].

A broader approach to further work in developing alternative MIDI data sources, such as using the OCZ Neural Impulse Actuator, could be worked on to provide a truly unique way of generating music with everyone from children to those with disabilities being able to use this system which would make for a really wide ranging device. Given that the cost of this device is only in the region of £100 it is also comparably inexpensive for the potential impact it could have.

7. ACKNOWLEDGEMENTS

This work was supported by Ben Shirley and Francis Li of the University of Salford. Initial work on MIDI interfacing library was produced in conjunction with Robert Kennedy at the University of Salford. The people who volunteered for the subjective testing are also highly valued as contributors toward this project.

8. REFERENCES

[1] Roland (UK) Ltd. (2009, April 10). *Roland UK - GI-10: GUITAR MIDI INTERFACE*. Retrieved April 10, 2009, from Roland UK: http://www.roland.co.uk/other_catdet.asp?id=GI10

[2] Yamaha Corporation of America. (2009, April 10). *WX5*. Retrieved April 10, 2009, from Yamaha Musical Instruments and Sound Reinforcement Products: http://www.yamaha.com/yamahavgn/CDA/ContentDetail/ModelSeriesDetail/0,6373,CNTID%253D1040%2526CTID%253D208500%2526VNM%253DLIVE%2526AFLG%253DY%2526LGFL%253DN,00.html

[3] Downes, P. (1987). Motion Sensing in Music and Dance Performance. *AES 5th International Conference: Music and Digital Technology*, (pp. 165-172).

[4] Bradshaw, D., & Ng, K. (2008). Tracking Conductors Hand Movements Using Multiple Wiimotes. *Automated solutions for Cross Media Content and Multi-channel Distribution, 2008. AXMEDIS '08. International Conference on* (pp. 93-99). Florence: IEEE.

[5] Hoggins, T. (2008, October 25). Wii: this time it's musical. *Daily Telegraph, The* , p. 20.

[6] Games Press Ltd. (2009, January 14). *Bastion Press Centre: 505 GAMES FEELS THE RHYTHM WITH WE ROCK: DRUM KING FOR NINTENDO WII™*. Retrieved April 10, 2009, from Bastion Press Centre: http://bastion.gamespress.com/release.asp?i=1510

[7] Jaroslow, G., & Roads, C. (1995). "Analog" Control of Digital Audio Signal Processing. *Journal of the Audio Engineering Society* .

[8] Warman, M. (2008, December 4). The final click of the mouse. *Daily Telegraph, The* , p. 26.

[9] Nintendo Co., Ltd. (2008). *Consolidated Financial Highlights.* Nintendo.

[10] Anderton, C. (1987). The MIDI Protocol. *5th International Conference: Music and Digital Technology.* Berkeley: Audio Engineering Society.

[11] Microsoft Corporation. (2009). *__fastcall (C++).* Retrieved April 10, 2009, from Visual C++ Developer Centre: http://msdn.microsoft.com/en-us/library/6xa169sk(VS.80).aspx

[12] Axelson, J. (2005). *USB Complete, Third Edition.* Madison: Lakeview Research LLC.

[13] WiiMote Project. (n.d.). *WiiMote Project*. Retrieved 05 09, 2009, from WiiMote Project: http://www.wiimoteproject.com/

[14] *Wiimote API's - Equis*. (n.d.). Retrieved April 10, 2009, from Equis: http://dundee.cs.queensu.ca/wiki/index.php/Wiimote_API%27s

[15] JVCL Team. (n.d.). *JVCL Home Page*. Retrieved April 10, 2009, from JEDI Visual Component Library: http://jvcl.delphi-jedi.org/

[16] Lee, J. C. (2008). Hacking the Nintendo Wii Remote. *Pervasive Computing, IEEE* , 39-45.

[17] *Motion Analysis*. (2009, January 18). Retrieved April 10, 2009, from Wiili. org Wii Linux: http://www.wiili.org/index.php/Motion_analysis

[18] *IR Sensor*. (2008, November 10). Retrieved April 10, 2009, from Wiimote Wiki: http://wiki.wiimoteproject.com/IR_Sensor

[19] Microsoft Corporation. (2009, April 9). *SetWaitableTimer Function (Windows)*. Retrieved April 10, 2009, from Microsoft Developer Network: http://msdn.microsoft.com/en-us/library/ms686289(VS.85).aspx

[20] Grey, J. M. (1975). *An exploration of musical timbre.* Stanford: Dept. of Music, Stanford University.

[21] Heise, S., Hlatky, M., & Loviscach, J. (2008). SoundTorch: Quick Browsing in Large Audio Collections. *Audio Engineering Society.* San Francisco: Audio Engineering Society.

[22] Ascii Art & Resources. (n.d.). *Ascii.ws - Ascii Chart, Ascii Table & Extended Ascii Table*. Retrieved April 10, 2009, from Ascii.ws: http://www.ascii.ws/ascii-chart.html

[23] Albin, S. T. (2003). *The Art of Software Architecture.* Indianapolis: Wiley Publishing, Inc.

[24] Crown, J. (1992). *Effective Computer User Documentation.* New York: Van Nostrand Reinhold.

[25] Schlömer, T., Poppinga, B., Henze, N., & Boll, S. (2008). Gesture Recognition with a Wii Controller. *Proceedings of the Second International Conference on Tangible and Embedded Interaction (TEI'08)* (pp. 11-14). Bonn: Association for Computing Machinery.

[26] Wang, Z., & Louey, J. (2008). Economical Solution for an Easy to Use Interactive Whiteboard. *Frontier of Computer Science and Technology, 2008. FCST '08. Japan-China Joint Workshop on* (pp. 197-203). Nagasahi: IEEE Computer Society.

[27] Wang, Y., Yu, T., Shi, L., & Li, Z. (2008). Using human body gestures as inputs for gaming via depth analysis. *Multimedia and Expo, 2008 IEEE International Conference on* (pp. 993-996). Hannover: IEEE.

[28] Rosenberg, C. S. (1994). Interface Design for Computer-Controlled Audio Systems. *13th International Conference* (pp. 41-45). Audio Engineering Society.

[29] AES, S. (2003). MIDI and Musical Instrument Control. *JAES Volume 51 Issue 4* , 272-276.

[30] Rosenberg, C., & Moses, B. (1993). Future Human Interfaces to Computer-Controlled Sound Systems. *95th Convention.* New York: Audio Engineering Society.

[31] Hartson, H. R. (1998). Human-computer interaction: interdisciplinary roots and trends. *Journal of Systems and Software* , 103-118.

[32] Hay, S., Newman, J., & Harle, R. (2008). Optical tracking using commodity hardware. *Mixed and Augmented Reality, 2008. ISMAR 2008. 7th IEEE/ACM International Symposium on* (pp. 159-160). Cambridge: IEEE.

[33] Vafadar, M., & Behrad, A. (2008). Human hand gesture recognition using spatio-temporal volumes for human-computer interaction. *Telecommunications, 2008. IST 2008. International Symposium on* (pp. 713-718). Tehran: IEEE.

[34] Cadoz, C., Luciani, A., & Florens, J.-L. (1984). Gesture, Instrument and Musical Creation: The System Anima/Cordis. *75th Convention.* Paris: Audio Engineering Society.

[35] Price, J. (1984). *How To Write A Computer Manual: A Handbook Of Software Documentation.* Menlo Park: The Benjamin/Cummings Publishing Company, Inc.

[36] Rumsey, F. (1994). *MIDI Systems and Control. 2nd Ed.* Oxford: Focal Press.

[37] Moxy, J. (2008, April 16). *Midi Message Format*. Retrieved April 10, 2009, from Songstuff: http://recording.songstuff.com/article/midi_message_format

[38] LiquidIce. (2006, Devember 3). *HOWTO: Use the Wii-Mote In Windows as your Mouse*. Retrieved April 10, 2009, from LiquidIce's Nintendo Wii Hacks: http://wiihacks.blogspot.com/2006/12/howto-use-wii-mote-in-windows-as-your.html

[39] *WiiMote*. (2009, March 29). Retrieved April 10, 2009, from WiiLi.org Wii Linux: http://www.wiili.org/Wiimote

[40] Brain Actuated Technologies, Inc. (n.d.). Retrieved April 10, 2009, from Brainfingers: http://www.brainfingers.com/

[41] USB Implementers Forum, Inc. (n.d.). *Tools*. Retrieved April 10, 2009, from USB.org: http://www.usb.org/developers/tools/

[42] Parise, J. (2002, December 28). *MIDI Input Library For Windows*. Retrieved April 10, 2009, from Computer Science House, Rochester Institute of Technology: http://www.csh.rit.edu/~jon/projects/midilib/

[43] EDais. (n.d.). *Using C++ DLL's in VB*. Retrieved April 10, 2009, from EDais: http://edais.mvps.org/Tutorials/CDLL/CDLLch1b.html

[44] Thie, M. (2008, August 29). *Marshalling: Using native DLLs in .NET*. Retrieved April 10, 2009, from rednael: http://blog.rednael.com/2008/08/29/MarshallingUsingNativeDLLsInNET.aspx

[45] Microsoft Corporation. (n.d.). *__stdcall (C++)*. Retrieved April 10, 2009, from Visual C++ Developer Center: http://msdn.microsoft.com/en-us/library/zxk0tw93(VS.80).aspx

[46] Verdone, M. (2007). *WiiToMidi - Wii Controller to MIDI interface for Mac OS X*. Retrieved April 12, 2009, from Mike Verdone: http://mike.verdone.ca/wiitomidi/

[47] WiiBrew. (2009, January 4). *WiiMote/Pointing*. Retrieved April 12, 2009, from WiiBrew.org: http://wiibrew.org/wiki/Wiimote/Pointing

[48] Dawn Of The Geeks. (2009, January 1). *A Better WiiMote Pointer*. Retrieved April 12, 2009, from Dawn Of The Geeks: http://blog.dawnofthegeeks.com/2009/01/01/a-better-wiimote-pointer/

[49] CondéNet, Inc. (2006, December 8). *Controlling a Synthesizer with the Wii Remote*. Retrieved April 12, 2009, from Wired Blogs: http://blog.wired.com/music/2006/12/controlling_a_s.html

[50] IVT Corporation. (n.d.). *BlueSoleil*. Retrieved April 12, 2009, from BlueSoleil: http://www.bluesoleil.com/products/index.asp?topic=bluesoleil6x

[51] Gorven, L. (2006, January 7). *General MIDI Standards: Table 2 - Percussion Key Map*. Retrieved April 12, 2009, from MIDIStudio: http://midistudio.com/Help/GMSpecs_PercMap.htm

9. APPENDIX

9.1. Project Plan

Project Plan

Title: BGS2G, Alternate MIDI Controller Implementation

Students: David Lewthwaite, Robert Kennedy

Supervisor: Ben Shirley

Second Reader: Francis Li

Introduction:

Using none-standard, alternative methods of inputting data into a computer and then using this to create and control MIDI messages. Most MIDI input systems are restricted to emulating traditional instruments (ie Keyboard). This project will investigate what other input devices could be used to make music.

Objectives:

Minimum Objectives:

- Investigate input systems available
- Produce MIDI interfacing DLL which includes SysEx data exchange
- Produce basic interface software to control midi messages eg
    - On screen keyboard
    - Program change
    - Effects control
- Ability to send raw SysEx directly from user interface
- Implement an alternative input system eg Nintendo Wii Controller, Joystick, 3D Mouse, OCZ Neural Impulse Actuator. The choice will be based on initial research and literature review as well as how achievable it is within the time frame.

Extended objectives:

- Extend user interface and investigate extended functions
- Test with test subjects who have a varying range of technical and/or musical ability.

Equipment Requirements:

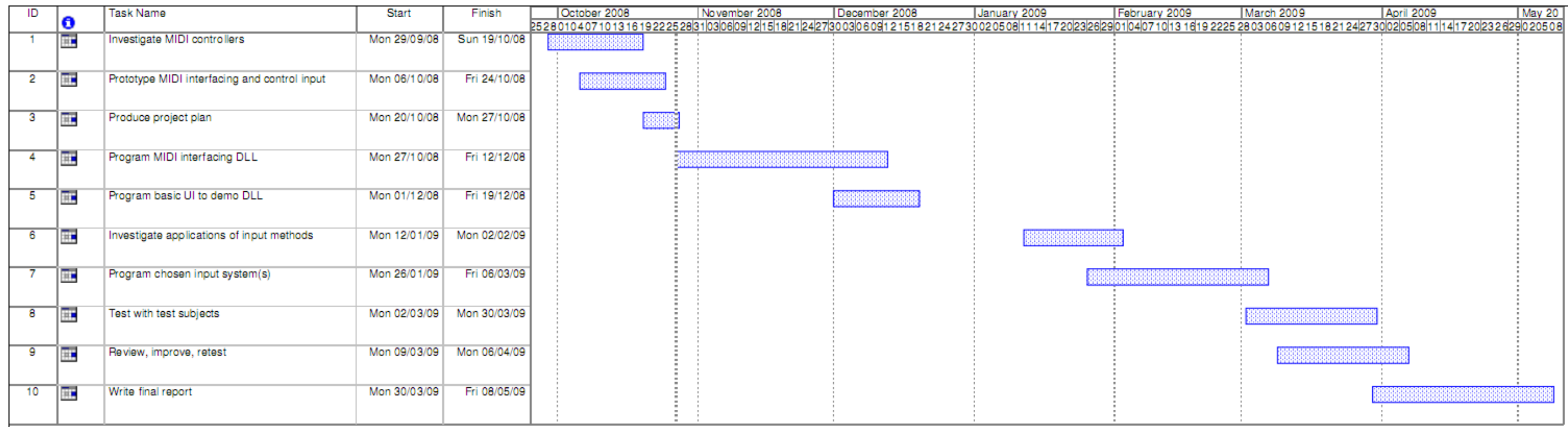
Depending on the objectives achieved:

- Computer for programming software
- MIDI Keyboard / Synth for testing complex data (ie SysEx) exchange
- Physical input device(s) – Wii Controller, OCZ Neural Impulse Actuator

Signature of Students:

Signature of Supervisor:

Signature of Second Reader:



ID	Task Name	Start	Finish
1	Investigate MIDI controllers	Mon 29/09/08	Sun 19/10/08
2	Prototype MIDI interfacing and control input	Mon 06/10/08	Fri 24/10/08
3	Produce project plan	Mon 20/10/08	Mon 27/10/08
4	Program MIDI interfacing DLL	Mon 27/10/08	Fri 12/12/08
5	Program basic UI to demo DLL	Mon 01/12/08	Fri 19/12/08
6	Investigate applications of input methods	Mon 12/01/09	Mon 02/02/09
7	Program chosen input system(s)	Mon 26/01/09	Fri 06/03/09
8	Test with test subjects	Mon 02/03/09	Mon 30/03/09
9	Review, improve, retest	Mon 09/03/09	Mon 06/04/09
10	Write final report	Mon 30/03/09	Fri 08/05/09

Figure 13, Project plan timeline

9.2. Explanation of Bit Shifting and OR Operator

Bit-shift decoding works as follows, for example, a 10-bit value is encoded across two bytes, byte 1 has the first 8 bits of the value (LSB), while byte two contains the final two bits (MSB), if these two bits are in positions 5 and 6, the retrieval of the full 10-bit value works as follows,

MSB byte, wanted bits located at positions 5 and 6,

```
1111 0101
```

Shift this down to the LSB, to zero unwanted bits

```
DataByte = DataByteMSB >> 4

1111 0101 >> 4 = 0000 1111
```

Shift back up to only have the wanted bits set

```
DataByte = DataByteMSB << 6

0000 1111 << 6 = 1100 0000
```

Place this into bits 9 and 10 of a new integer value (or any 10-bit or greater value)

```
int HorizPos = DataByteMSB << 2

1100 0000 << 2 = 0011 0000 0000
```

(Only first 12 bits shown)

The LSB byte can then be combined with this using the OR operator which sets an output bit to ‘1’ if either of the input bits are ‘1’,

```
HorizPos |= DataByteLSB

0011 0000 0000 OR 0101 1100 = 0011 0101 1100
```

This 10-bit output value can then be scaled into a floating point decimal for use as a fraction or percentage.

9.3. Explanation of Class Inheritance

Class inheritance is an incredibly useful part of object orientated programming. It allows for future classes to build on others. For example, a base class could contain a function for sending MIDI SysEx data. A class inherited from this would have access to this function (as well as any others) without having to write a specific implementation. The new class can then have individual functions for sending specific SysEx messages, internally calling the SysEx function with the raw message.

More broadly, a class to represent a car could inherit from a class for vehicles – the car automatically inherits all properties of vehicles – they have engines, passengers, wheels and so on. The car implements the functions and values in its' own way so that they apply specifically to a car and not to for example, a lorry or motorbike. A car could have extra functions for added options such as electric windows or satellite navigation where obviously neither of these would be applicable to a motorbike.

9.4. ASCII Chart

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Figure 14: ASCII table [22]

9.5. Explanation of Bit-Masking and Bitwise AND Operator

Decoding bit-set values from a two-byte message into independent Boolean status values, is simply a matter of performing the bitwise AND operator with the data byte and the defined bit mask values. For example, the status of the 'left' button on the WiiMote is defined at bit `0` of byte `0`, this has a bitmask of `0x01`. A hex value of `0x01` will set bit `0` of a byte as this will represent the number '`1`' as follows:

```
0000 0001 = 0x01
```

If the status of the left button is 'pressed' (i.e., it is being held down) then bit 0 of the message will be set to 1, where the rest of the bits can be either 1 or 0. The bitwise AND operator will combine the two bytes, and where both corresponding bits are '1', will set the output bit to '1' or '0' otherwise. Performing the AND operator on the example gives the following,

```
Bitmask & Data =

0000 00001 & 0100 0111 = 0000 0001
```

So the output is a positive value, so is equal to Boolean true.

9.6. Explanation of Function Pointers in Borland

In traditional 'C' code, function pointers are generally referred to as callbacks, these are where a class can literally 'call back' into higher level code, usually with incoming data. In diagram form, this would look like a loop,

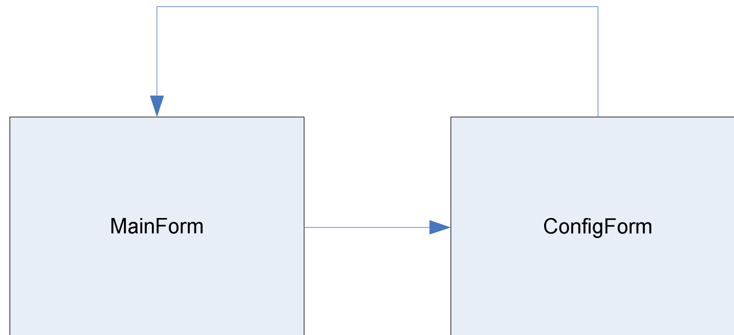


Figure 15: Callback example

In normal circumstances, the `MainForm` class creates and calls functions within `ConfigForm`. Some action within `ConfigForm` uses a 'callback' to call into `MainForm`. Borland makes defining callbacks easy in the form of `__closure` definitions. For example, `ConfigForm` would have the following function defined above its class definition,

```
typedef void __fastcall( __closure *fnCallBack)(int aId);
```

This defines a new type which the application can use, within the class definition for `ConfigForm` an instance of this is created,

```
fnCallBack OnCallBack
```

The 'On' prefix is simply a convention to provide easy naming as callbacks are generally event based. In this case it would occur when the user made a specific action.

In order to avoid access violation errors, within the constructor of `ConfigForm` the function pointer is set to NULL,

```
OnCallBack = NULL
```

The creating class, `MainForm`, attaches the real function to the pointer, so within `MainForm` there is a function which has the definition

```
void __fastcall MainForm::CallBackFunction(int aId)
```

The definition of the function to be used as the callback must be the same as the function pointer definition or the application will fail to compile. After creating `ConfigForm`, `MainForm` will set the value of the function pointer,

```
ConfigForm *Conf = new ConfigForm();
Conf->OnCallBack = CallBackFunction;
```

Now whenever `ConfigForm` calls the function `OnCallBack` it will execute as a function within `MainForm`. To prevent access violation errors, before any call of `OnCallBack`, a simple if test to check if the pointer is valid is all that is required, since the original function was set to NULL during the constructor. To that end, all function calls from `ConfigForm` look like this,

```
If(OnCallBack)
 OnCallBack(Number);
```

(Where ‘Number’ is an integer value).

9.7. User Testing Questionnaire

Subjective Testing for WiiMote-MIDI Interface System

Thank you for taking part in the subjective tests for the WiiMote-MIDI Interface. You will be asked to listen to the computer playing back selected drum patterns and attempt to follow them using the WiiMotes. The tests are completely anonymous but there will be two short questionnaires to complete before and after the tests.

You will now be given some time to familiarise yourself with the system. The WiiMotes have already been connected and calibrated so simply point the WiiMote at the screen to move the virtual drum sticks about. The drum will trigger when the WiiMote is moved downwards, into the coloured area. The system is velocity sensitive, so it may take a while to get used to.

The computer will play back 10 different drum patterns, in a range of difficulties. Your aim is to match the pattern being played by the computer. Take as much time as you need. When you are ready to move into the next pattern, press the B button on the back of the WiiMote.

The computer will assess you based on the following:

- Time Taken to get the pattern correct
- Accuracy of notes (both in relation time and correct drum sound)
- Velocity consistency

When you are ready, the tests will begin.

Signed:

Date:

Subject No:

Pre Test Questionnaire:

1. Do you play a musical instrument? (Please specify)

2. If you answered yes to (1), how would you rate your musical ability (on a scale of 1-10)?

3. Do you play computer games? If so which genre(s)?

4. Have you ever played on a Nintendo Wii?

Post Test Questionnaire

1. Was the system easy to use (on a scale of 1-10)?

2. Where could you see such a system (or the technology within) being used?

3. What did you find difficult or unintuitive about the system?

9.8. User Testing Results Tables

Test Subject 1					
Pattern	Notes In Pattern	On Target	Wrong Notes	Missed Notes	Percent Correct
0	31	11	560	20	35.48
1	55	23	0	32	41.82
2	67	40	0	27	59.70
3	126	46	0	80	36.51
4	109	23	0	86	21.10
5	119	43	0	77	36.13
6	130	25	0	105	19.23
7	232	48	0	184	20.69
8	130	40	0	92	30.77
9	110	8	0	102	7.27
Standard Deviation of Velocity			26.806781		

Table 6: Test subject 1 results

Test Subject 2					
Pattern	Notes In Pattern	On Target	Wrong Notes	Missed Notes	Percent Correct
0	52	42	887	10	80.77
1	67	50	0	17	74.63
2	88	59	0	29	67.05
3	82	57	0	25	69.51
4	105	74	0	31	70.48
5	279	164	0	128	58.78
6	321	120	0	209	37.38
7	444	79	0	366	17.79
8	215	154	0	83	71.63
9	590	271	0	346	45.93
Standard Deviation of Velocity			23.444343		

Table 7: Test subject 2 results

Test Subject 3					
Pattern	Notes In Pattern	On Target	Wrong Notes	Missed Notes	Percent Correct
0	22	16	733	6	72.73
1	70	21	34	49	30.00
2	61	32	11	29	52.46
3	61	5	6	56	8.20
4	81	61	13	20	75.31
5	73	31	7	42	42.47
6	61	14	4	47	22.95
7	136	10	7	126	7.35
8	152	96	66	66	63.16
9	318	111	33	221	34.91
Standard Deviation of Velocity			29.581011		

Table 8: Test subject 3 results

Test Subject 4					
Pattern	Notes In Pattern	On Target	Wrong Notes	Missed Notes	Percent Correct
0	28	18	137	10	64.29
1	50	25	1	25	50.00
2	32	21	0	11	65.63
3	32	12	1	20	37.50
4	39	13	0	26	33.33
5	120	37	13	83	30.83
6	7	0	0	7	0.00
7	238	56	14	182	23.53
8	97	55	3	49	56.70
9	230	72	6	160	31.30
Standard Deviation of Velocity			22.102827		

Table 9: Test subject 4 results

Test Subject 5					
Pattern	Notes In Pattern	On Target	Wrong Notes	Missed Notes	Percent Correct
0	18	13	334	5	72.22
1	26	15	12	11	57.69
2	54	25	21	29	46.30
3	34	14	1	20	41.18
4	35	16	1	19	45.71
5	49	15	1	34	30.61
6	90	20	2	71	22.22
7	88	16	0	72	18.18
8	91	51	6	46	56.04
9	175	30	5	146	17.14
Standard Deviation of Velocity			21.420181		

Table 10: Test subject 5 results

Test Subject 6					
Pattern	Notes In Pattern	On Target	Wrong Notes	Missed Notes	Percent Correct
0	53	14	59	39	26.42
1	3	0	0	3	0.00
2	30	7	1	23	23.33
3	21	5	0	16	23.81
4	33	3	0	30	9.09
5	48	6	7	43	12.50
6	27	3	2	24	11.11
7	76	13	5	63	17.11
8	68	29	5	40	42.65
9	133	26	15	108	19.55
Standard Deviation of Velocity			26.457885		

Table 11: Test subject 6 results

Test Subject 7					
Pattern	Notes In Pattern	On Target	Wrong Notes	Missed Notes	Percent Correct
0	20	16	3	4	80.00
1	47	21	17	26	44.68
2	31	11	16	20	35.48
3	50	22	16	28	44.00
4	42	20	3	22	47.62
5	32	9	3	23	28.13
6	38	7	6	31	18.42
7	36	0	5	36	0.00
8	75	45	13	32	60.00
9	54	17	8	38	31.48
Standard Deviation of Velocity			32.789179		

Table 12: Test subject 7 results

9.9. Accompanying CD-ROM

The accompanying CD-ROM contains both the source code and PDF edition of this written report. The directory structure is as follows:

/ModernMIDI.pdf
Electronic copy of this report.

/MidiInterface
MIDI DLL.

/MidiTestApplication
Application created to demonstrate and test the MIDI Interface.

/ResultsAnalyser
Application used to analyser the results of user testing.

/UnitTests
DUnit tests for technical testing of classes.

/WiiMoteDrums
The main application produced by this project.