



HMS-1 Hybrid Musical Synthesizer



Project Report

Ben Davey

Dedicated to the legacy of Tristram Cary (1925-2008). Electronic music pioneer.

Executive Summary

Broadly, the purpose of this project was to create an *Analogue modelling musical synthesizer*; that is, a synthesizer implemented using both analogue and digital components that intends to emulate the sounds of traditional analogue synthesizers, whilst being controllable through modern-day digital protocols such as the Musical Instrument Digital Interface (MIDI).

The synthesizer platform exists upon it's own standalone hardware, and may be controlled using a variety of existing MIDI-capable instruments, including keyboards, guitars and drums. Additionally, the unit features it's own innovative touch-screen interface which intends to provide simple-and-intuitive control of the system, but with an unprecedented ability to customize and control synthesizer parameters and sound generation.

Special attention has been placed upon the usability of the synthesizer, particularly within the realms of it being played by musicians in a live environment.

The unit's MIDI capabilities also mean that it is easily combined with professional studio equipment and sequencing software such as Cubase VST

Contents

Introduction.....	6
Purpose & Scope.....	6
Scope.....	6
Abbreviations and Acronyms.....	6
Features at a glance.....	7
System Overview.....	10
MIDI Input.....	12
Protocol Overview.....	12
Built-in surface.....	12
Overview	12
Control Logic.....	12
Rotary Knob configuration.....	13
MIDI output.....	13
Circuit Layout.....	14
MIDI Interfaces.....	15
Device Descriptors.....	15
Software Sequencer Mixing.....	16
Synth Controller.....	17
Control Hub overview.....	17
Hardware overview.....	17
Configuration and kernel.....	18
Installed packages.....	19
Package management tools.....	19
other packages.....	19
Software platform overview.....	21
Python.....	21
MIDI Interpreter.....	22
Overview.....	22
Item controllers.....	23
GUI Widgets.....	25

Main Interpreter.....	27
Event flow	28
Attributes.....	28
Methods.....	29
Timing example – Note-on event.....	30
Arpeggiator.....	32
Technical overview.....	32
Connection to the MIDI interpreter.....	33
Algorithm.....	33
Envelope Generator.....	34
Technical overview.....	34
Drawing surface.....	35
Sample acquisition.....	35
Sample scaling and transmission.....	36
Graphical User Interface.....	37
Technical overview.....	37
Development Process.....	37
Widgets.....	38
Widgets implemented.....	39
Common methods.....	39
Panels.....	41
Root canvas/ Main window.....	42
Project case.....	43
Time management.....	45
Milestones.....	45
Conclusion.....	52
References.....	54

Introduction

Purpose & Scope

This document provides a reference for the architecture and design techniques employed in the development of the Hybrid Musical Synthesizer (HMS) developed by Synthesia between March – October 2008. It is intended to provide an overview of the system, as well as functional specifications of its constituent components.

Scope

This document provides an overview of the design techniques, technologies, and implementation methods of the parts of the synthesizer that I was instrumental in creating for the project. Overviews of other system components may be found in the reports of Nathan Sweetman, Andreas Wulf, and Aaron Frishling.

Abbreviations and Acronyms

ADSR	-	Attack, Decay, Sustain & Release
COTS	-	Commercial Off The Shelf
DAC	-	Digital to Analogue Convertor
FPGA	-	Field Programmable Gate Array
GUI	-	Graphical User Interface
LCD	-	Liquid Crystal Display
MIDI	-	Musical Instrument Digital Interface
VHDL	-	VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
I/O	-	Input/Output
SVG	-	Scalable Vector Graphics - an open architecture vector graphics format
PNG	-	Portable Network Graphics - an open architecture bitmap graphics format
DMA	-	Direct Memory Access
LFO	-	Low Frequency Oscillator

Features at a glance

In its final form the project implemented the following features:

- **MIDI input from a variety of instruments**

Unlike traditional analogue synthesizers, whose input was limited to a keyboard, ribbon controller, or patching area, the HMS-1 is compliant with general MIDI specifications. This means that any MIDI-enabled device can be used to operate the device, including guitars, sequencers, flutes and keyboards.

- **Fully assignable controls**

The HMS-1 has the ability to have nearly all controls (excluding envelope drawing) automated over MIDI. This enables the parameters to be altered using tactile MIDI control surfaces or even sequenced and time-automated using MIDI scoring software.

- **Tactile control using built-in surface**

For increased usability in a live environment, the synthesizer features it's own 8-knob tactile control surface. The knobs included are highly sensitive and are fully assignable to synth parameters.

- **Intuitive touch screen user interface**

The synthesizer was developed with the needs of musicians in mind, and an elegant graphical user interface was developed that is powerful, yet easy to use was implemented and is easily navigated and controlled by a musician's finger.

- **Vocoder/ Ring modulator**

The HMS-1 features a line input who's signal can be modulated and filtered with that of the internal oscillators, allowing it to be connected to an external hardware such as a function generator or microphone for ring-modulation or vocoder-type effects.

- **Selectable note-priority scheme**

Due to its monophonic nature, the synthesizer features the ability to select which priority scheme is used to set its resulting output note frequency.

- **Monophonic Arpeggiator**

A monophonic arpeggiator based loosely on that found in the Roland Jupiter 4 has been included as part of the synthesizer's firmware. This supports the following arp patterns:

- Up
- Down
- Up+Down
- Random

- **3 independent oscillators with customisable offsets and waveforms**

The HMS-1 features 3 independent oscillators; the frequency of each can be offset by +/- 1 octave from the input note. These oscillators have settable frequency between 1 and 25088Hz and can be set to the following periodic waveforms:

- Sine
- Square
- Triangle
- Sawtooth

- **Low Frequency Oscillator (LFO)**

An LFO has been included with the system to control the vibrato of output sound. Its output frequency can be set between 0-20Hz and is modulated with the output of the

other oscillators

- **Selection of oscillator mixing styles**

The synthesizer features the ability to select between additive and multiplicative mixing techniques for its oscillator outputs. Subtractive mixing may be considered in future revisions of the project

- **Software driven legato and portamento control**

The HMS-1's onboard support for legato and portamento allows for note-holding and smooth note transitions at an adjustable rate

- **Adjustable, digitally controlled analogue filter and resonator**

At the heart of its analogue post-processing functionality, the synthesizer features a digitally controlled CEM3389 analogue signal processor. This provides the synthesizer with a wide-range, rich sounding Voltage Controlled Filter (VCF) with adjustable resonance and cutoff frequency parameters.

- **Novel touch-gesture based envelope generator**

The synthesizer's note amplitude envelope can be "drawn in" with the user's finger, creating a highly original experience in synth control

- **Ability to save and load preset patches**

MIDI mappings and parameter settings may be loaded and saved to file and shared amongst users in the synth community using the synthesizer's preset functionality. The interface features an onscreen keyboard for the naming of preset patches, and the device has enough user-dedicated memory to support a near-infinite amount of presets.

System Overview

The system was designed in modules according to Figure 1. Its main components were grouped into functional blocks:

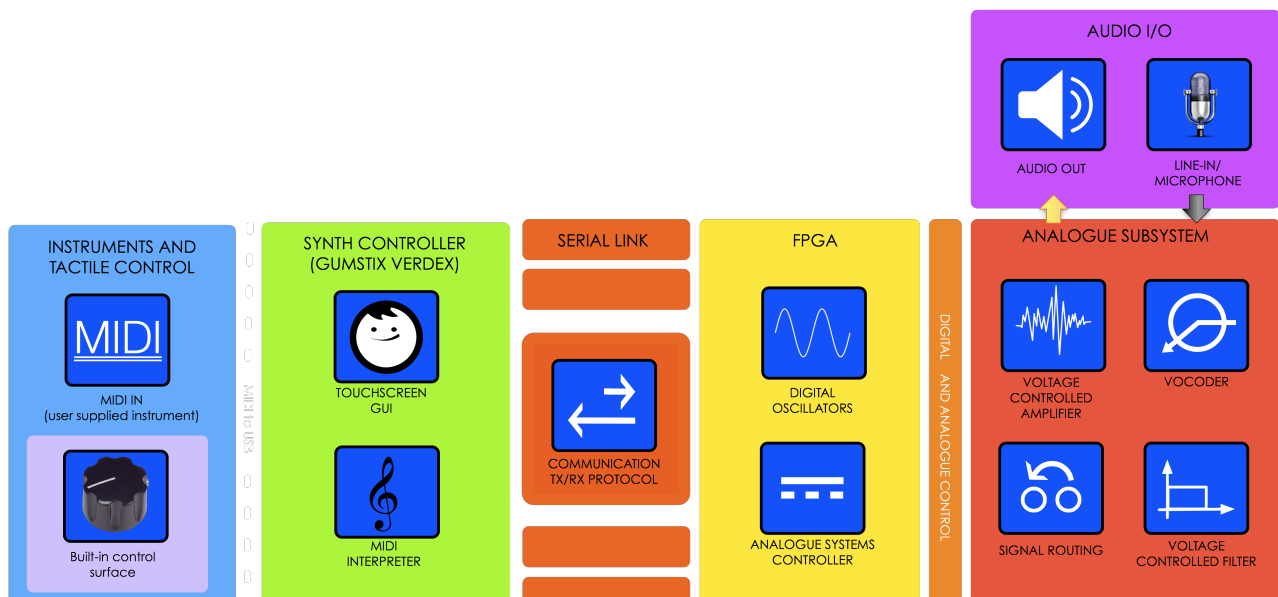


Figure 1. Block diagram of the system

MIDI Input

Allows MIDI compliant devices to be connected to the system using a standard 5-pin DIN connection. Additionally, this is where the systems inbuilt control surface is connected.

Synth Controller

Interprets MIDI input and provides the system's touch-screen user interface. Its key role is to convert MIDI input into system control signals and graphical events pertaining to the synth's

audible functions, and distribute them across the serial link to to the relevant synth module (e.g. conversion of MIDI note signals into to frequency control vectors to be sent to the FPGA). The synth controller also provides the high level, time-based audio control functionally. It is responsible for effects such as Envelope generation,

Serial Link

The serial link connects the Synth Controller and the FPGA. It consists of a physical connection between the devices, as well as an optimised communications protocol to enable the setting of synth parameters such as oscillator frequencies, and analogue systems control parameters.

FPGA

The heart of audio control systems and sound generation. The FPGA is responsible for oscillator function generation, mixing, and basic audio synthesis in the digital domain. It also provides digital control for settable analogue parameters, as well as digital-to-analogue conversion of the audio signal prior to analogue processing.

Analogue Systems

Provides core audio post-production functionality and is controlled by the FPGA. The analogue systems consists of various mixers, ICs, and DACs (for audio and control signals. It's key components are a voltage controlled filter, a voltage controlled amplifier, and a vocoder.

MIDI Input

Protocol Overview

MIDI (Musical Instrument Digital Interface) is an industry-standard protocol that enables electronic musical instruments, computers, and other equipment to communicate, control, and synchronize with each other. MIDI allows computers, synthesizers, MIDI controllers, sound cards, samplers and drum machines to control one another, and to exchange system data.¹

MIDI messages are sent serially, and are organized into 10-bit words. Most messages usually contain between 3-4 words, with the first word indicating the type of control signal (eg note-on, note-off, control change, etc) and intended destination “channel”, whilst the next words contain parameter values for these messages.

Our synthesizer is capable of interpreting note-on, note-off, control-change and pitch-bend messages. All other messages are ignored by the interpreter.

Built-in surface

Overview

A built-in control surface was provided by Damien Poirer of Audio Visual Services. It is designed to operate on any MIDI channel between 1-16 (programmable via rudimentary interface), and has support for up to 8 analogue controls (mixer sliders or rotary pots), 8 status LEDs, and 8 dual-state buttons. It is also capable of being interfaced to an 20x2 LCD to provide debugging information.

Control Logic

¹ 1. Wikipedia – Musical Instrument Digital Interface - <http://en.wikipedia.org/wiki/MIDI>

The device's control logic has been implemented upon a PIC16877F micro-controller. This features a high quality 10-bit digital to analogue converter from which slider values are calculated

Rotary Knob configuration

All rotary knobs are 10kohm linear pots, whose terminals are connected directly to +5V and 0 respectively. Wipe output is connected into the provided header pins where it is processed by the provided control logic

MIDI output

The PIC's UART is connected to an opto-coupling device in order to produce standard MIDI output. At +5V.

Circuit Layout

Figure 2 below shows the circuit for the built-in controller as it was provided to us.

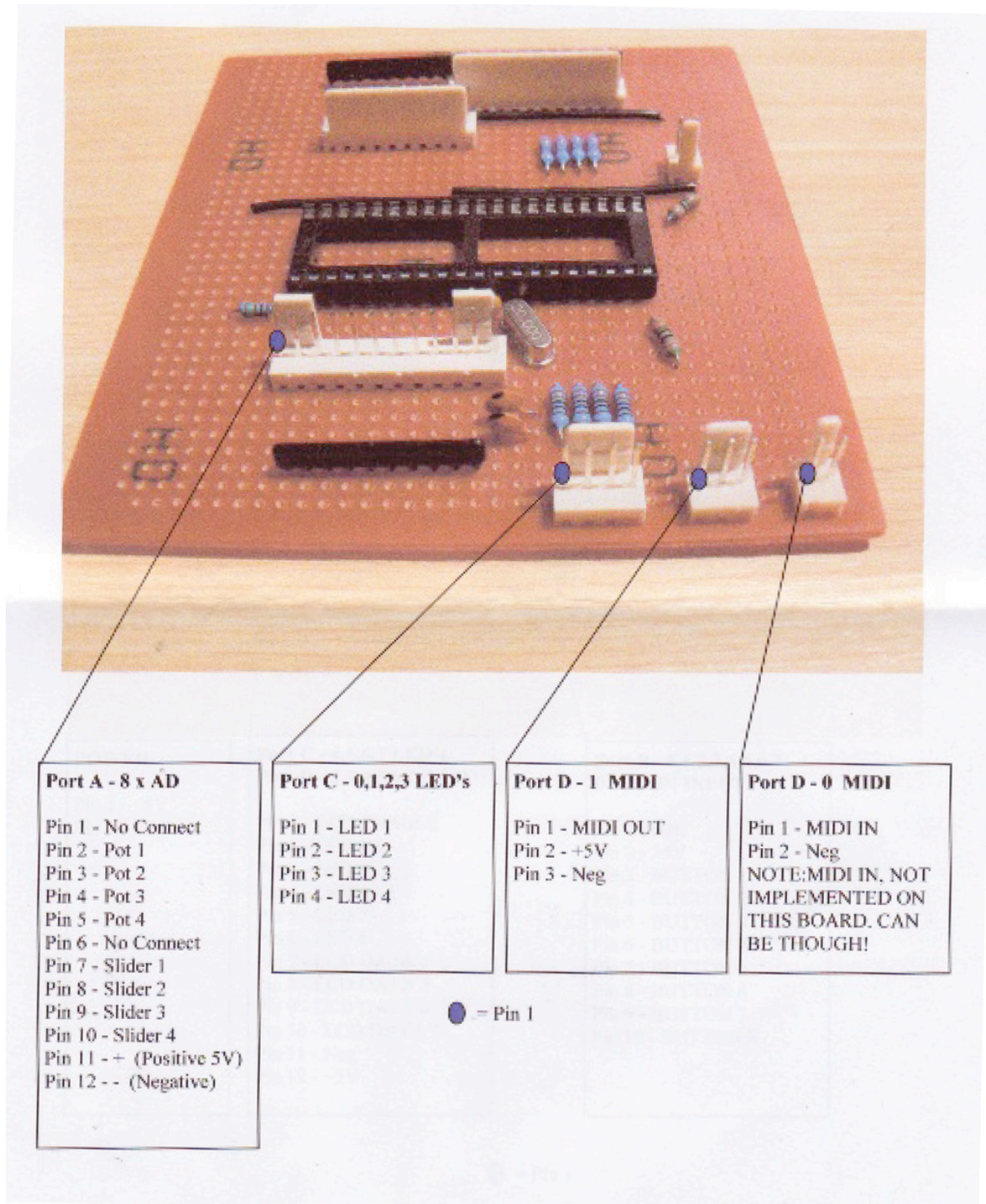


Figure 2 - Circuit layout of the built-in control surface

MIDI Interfaces

MIDI input is provided to the system over USB. This was achieved by purchasing 2 prefabricated USB->MIDI adapters:

1. **Roland UM-1 USB MIDI:**

Provides a reliable and robust interface with any MIDI input, and is wired to connect to a standard 5-pin DIN input port. This device was chosen because of Roland's strict coherence to MIDI specifications, and their use of high quality optocouplers and interface ICs.

2. **Generic USB MIDI:**

This device operates provides connection to the systems inbuilt control surface. It is similar to the UM-1 but is much less robust and less likely to strictly conform to MIDI specifications

Initially 2 generic interfaces were purchased, however we found one of them to have a high amount of latency and the tendency to sporadically 'miss' bytes from the MIDI messages it sent, which were then further misinterpreted by our software MIDI interpreter

Device Descriptors

In the linux environment, the devices were accessible using raw file descriptor nodes. These were statically defined at:

`/dev/snd/midic0p0` – Raw input from built-in control surface

`/dev/snd/midic1p0` – Input from UM-1 (MIDI in from keyboard or external device)

Software Sequencer Mixing

In order to minimize the complexity of our MIDI interpreting software, the 2 MIDI interfaces are mixed into a single stream using the operating system.

This was achieved using the Advanced Linux Sound Architecture (ALSA) user-space application “`aconnect`” (a utility to connect and disconnect two existing ports on the ALSA sequencer system) , as well as the loading of a virtual MIDI kernel extension, which provides dummy raw file descriptor and sequencer support to ALSA.

Using these applications we were able to redirect the output of both `/dev/snd/midiC0D0` and `/dev/snd/midiC1D0` to a universal, mixed descriptor, `/dev/snd/midiC2D0`.

Synth Controller

Control Hub overview

The control hub is the center of operations for all synthesizer control. It interprets MIDI input, registers control mappings, generates envelopes and arpeggios, and drives the user interface. It is responsible for setting oscillator frequencies, and instructing the FPGA on general control parameters.

Hardware overview

The control hub was implemented upon a Gumstix Verdex 5e embedded computer. This is a system approximately the size of a piece of chewing gum, but with near-desktop processing capabilities, infinitely customizable software options, and adequate IO capabilities.

It's key specifications are:

- ARM-5te processor @ 600MHz
- 32MB onboard flash memory, 64MB RAM
- USB host support (allows the device to act as a host to USB devices such as flash disks and MIDI interfaces)
- Expandable through daughterboards. We purchased several to provide the following:
 1. Gumstix NetmicroSD – Provides Ethernet and solid-state storage using a microSD card
 2. Gumstix ConsoleLCD – Provides 3 serial I/O ports and a 4.5" full-colour Samsung touchscreen enabled LCD.

A 1gb microSD card was also purchased for additional storage capabilities.

Configuration and kernel

Upon delivery, the gumstix arrived with no software (even a basic OS), and thus an operating system had to be prepared for it, and flashed onto it's memory using the Kermit serial transfer protocol.

Linux 2.6 kernel

We chose to cross-compile a predominantly monolithic Linux 2.6 kernel, patched with hardware support for the provided gumstix daughterboard's Ethernet, USB host controller, SD card-reader, and LCD frame buffer chipsets. ALSA sequencer support was also compiled as a module, as well as virtual MIDI support (snd_virmidi) and support for all interconnected devices. The list of compiled modules is too long for this overview document, however, configuration makefiles can be located in our SVN repository.

A customized bootsplash image containing the Synthesia logo was also implemented as part of the kernel compilation process.

Anagstrøm distribution & tools

The gumstix user community have created Anagstrøm, a modified version of the OpenMoko linux distribution for use on their platform. OpenMoko was created as an open-source operating platform targeted towards mobile phones. It provides the gumstix with basic package management tools, and a simplified and streamlined configuration.

User space applications on SD card

Finding the 32mb of available memory rather insufficient for our purposes, we altered kernel module loading files and initialization scripts to point our /usr directory to storage on the SD card. This is where all installed packages resided, and provided a convenient means of being able to backup or project's progress and the gumstix' configuration as the project progressed.

Installed packages

Package management tools

The gumstix SDK provides several applications that assisted with the streamlined installation of packages required:

- **The Itsy Package Management System (ipkg)** - a lightweight package management system designed for embedded devices that tries to resemble Debian's dpkg. This application is run upon the device itself, and allows precompiled packages to be readily installed from package repositories over the internet
- **BitBake** - a tool for the execution of tasks. It is derived from Portage, which is the package management system used by the Gentoo Linux distribution. Bitbake exists on the host platform (ie our development machines and not the gumstix itself), and enables relatively simplified cross-architecture compilation through the use of “recipes” – files written by us that define how to build a particular package. They include all the package dependencies, sources to fetch the source code from, configuration, compilation, build, install and remove instructions.

other packages

The following 3rd party applications and libraries were installed onto the gumstix using either ipkg or bitbake:

- **Python 2.5** – Interpreter for applications written in the python programming language (discussed later)
- **Nano-X window system** – An open source, cut down version of the X11 window system, targeted towards embedded devices. The project aims to deliver the features of modern graphical windowing environments to smaller devices and platforms²

| ² <http://en.wikipedia.org/wiki/Microwindows>

- Dropbear SSH daemon** – Provides a fully featured secure-shell implementation for use on embedded platforms. This was used to remote control the system over the network, and to transfer and deploy files from development machines to the synth controller.
- Glib** - Provides a portable object system and transparent cross-language interoperability. It was used fundamentally for its messaging and event driven capabilities
- GTK+** - a cross-platform widget toolkit for creating graphical user interfaces
- PyGTK** – python language bindings to access Glib and GTK+ functionality
- Pango** - library for rendering internationalized texts in high quality.
- Cairo** - A library used to provide a vector graphics-based, device-independent API for software developers. It is designed to provide primitives for 2-dimensional drawing across a number of different backends³
- Goocanvas** – A library intended to provide high level abstraction for drawing. Enables simple animation, grouping of objects, object clipping and other 2D graphic related features
- PySerial** – Library providing basic raw serial I/O functionality to the python programming language
- PyrtMIDI** – Library providing a higher level abstraction from raw MIDI I/O. Provides rudimentary interpretation of MIDI status bytes, and provides a simple object oriented architecture for their MIDI event handling and processing

³ http://en.wikipedia.org/wiki/Cairo_graphics_library

Software platform overview

Where possible, our team made use of existing and open source libraries. This was primarily so that we could focus on the main task at hand of developing a synthesizer, whilst having all the functionality we require.

Python

Python was chosen as the language upon which to develop all our software on the control hub. It is a high-level, interpreted programming language with minimalistic syntax. It was chosen for use in the system for the following reasons:

1. Object Orientation
2. Open source and cross platform
3. Highly forgiving and easy to prototype in
4. Binds easily to low level C libraries
5. Huge collection of available interfacing libraries

MIDI Interpreter

Overview

Figure 3 shows the architecture of the MIDI interpretation system. The MIDI interpreter connects to raw MIDI input from the operating system, and in-turn this connects it to the rest of the system. It communicates, and is interconnected with the graphical user interface and communications layers through “item controllers”- objects used to store and handle events and bindings

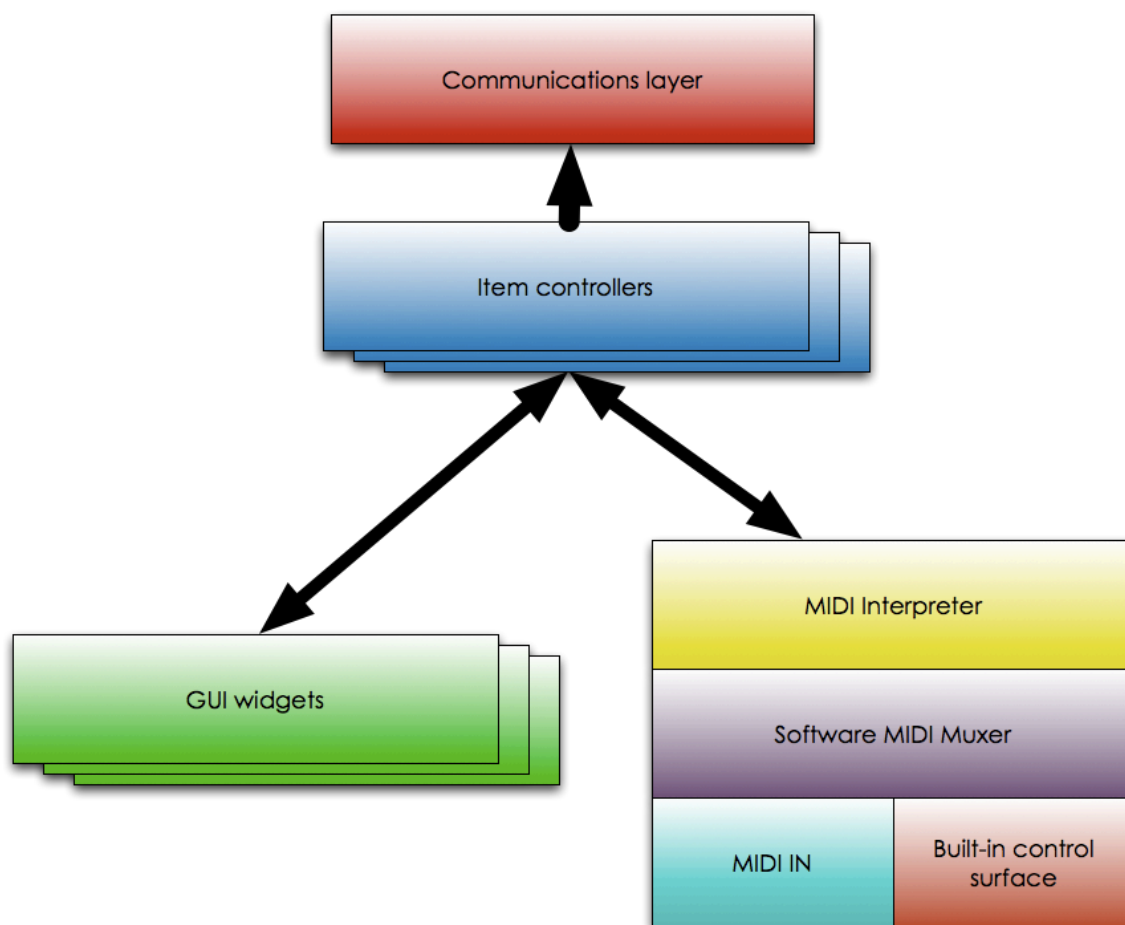


Figure 3- Architectural model of the synthesizer control hub

Item controllers

Item controllers are persistent data objects used to store parameter values (e.g. oscillator offsets, etc) and handle, translate and connect both GUI and MIDI events. Their operations are described below

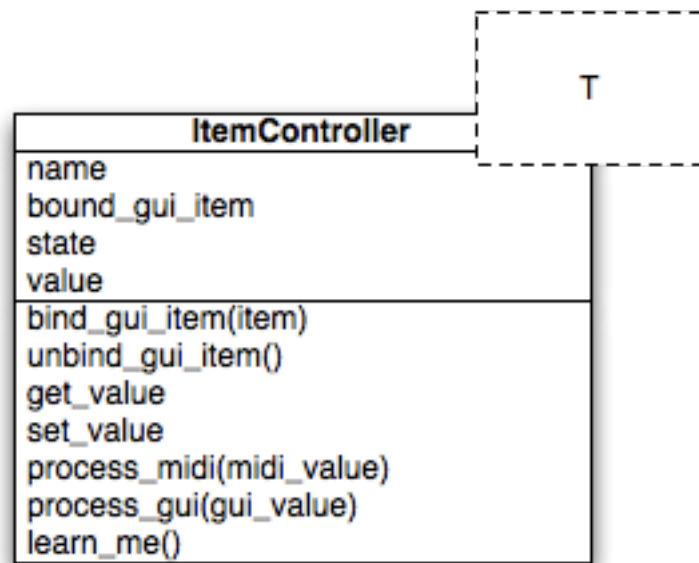


Figure 4- Class model of an item controller

Bind_gui_item(item)

Links the specified widget to the item controller. Binding means that any changes made to the widgets value will also change the value of the item controller and vice versa.

unbind_gui_item()

Unlinks any connected GUI widget from the controller. This method is usually invoked when the widget is destroyed or cannot be seen. Essentially this disconnects all event listeners from the GUI widget, and the controller will not update it on the event of a change of value

set_value(value)

Sets the value of the item controller. This method is overridden for each implementation of the item controller and connects to additional logic such as the communications layer to permeate changes in the controllers value to the FPGA

get_value()

Gets the current value of the item controller. This method is usually overridden by subclassed instances of the item controller.

process_midi(midi_value)

An adaptive method to set the value of a given item controller from a MIDI control message (ie 0..127). This method is usually called by the MIDI interpreter when the given item controller is registered to respond to a particular MIDI control event

process_gui()

An adaptive method to set the value of a given item controller from a GUI “value changed” signal. This method is usually called as an event responder to a GUI widget

learn_me()

Usually invoked from a GUI widget “learn me” signal. This method tells the main MIDI interpreter to register the item controller’s process_midi() responder to next incoming MIDI control event

GUI Widgets

GUI widgets are connected to the system through their intermediate item controllers. All GUI widgets conform to the class specification shown in Figure 5.

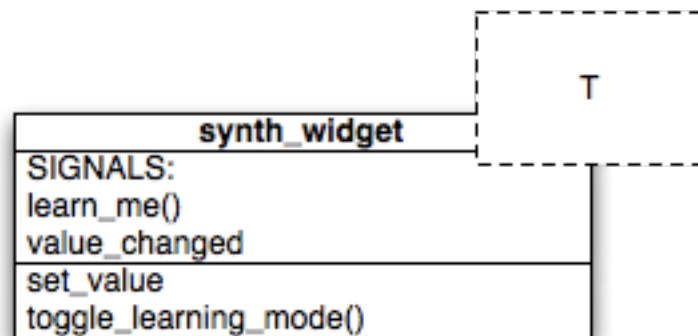


Figure 5 – Class overview of all GUI widgets

Depending on whether or not a widget is in learning mode, it will emit different event signals to its corresponding item controller. This is shown in the finite state diagram in figure 6:

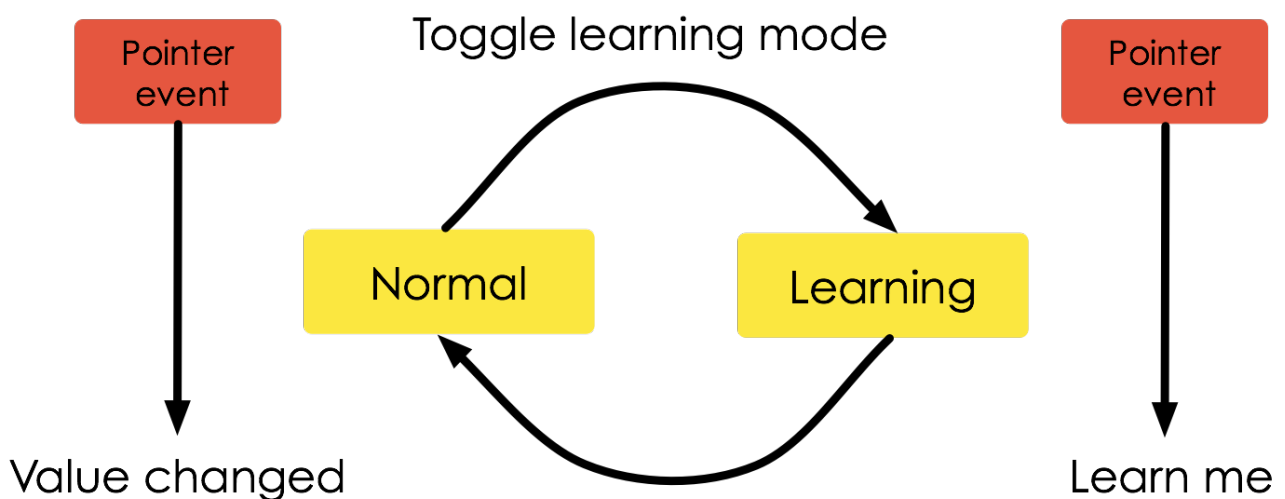


Figure 6 – states and signals emitted by GUI widgets

Learn_me

A “learn me” signal is emitted by the GUI widget when it is in learning mode. This is usually responded-to by a widget’s item controller, which then passes the event up to the MIDI interpreter to register the controller to a given GUI event

Value_changed

This signal is emitted when a control is manipulated in the GUI. It is usually connected to it’s given item controllers’ **process_gui** method.

Toggle_learning_mode

Turns the widgets learning mode Boolean either on or off, and changes the widget’s behavior and emitted event signals accordingly

Main Interpreter

The main interpreter listens for events, and processes them (either directly or through passing them to item controllers)

It houses the “preference classes” which are essentially collections of item controllers grouped into functional units, and is responsible for:

- Processing note events from the MIDI device and calculating their priority
- Processing and routing control change events to item controllers
- Performing pitch-bend transforms in realtime based on instrument input
- Triggering note-based effects such as the envelope generator and arpeggiator

The main interpreter's class diagram is shown below in figure 7.

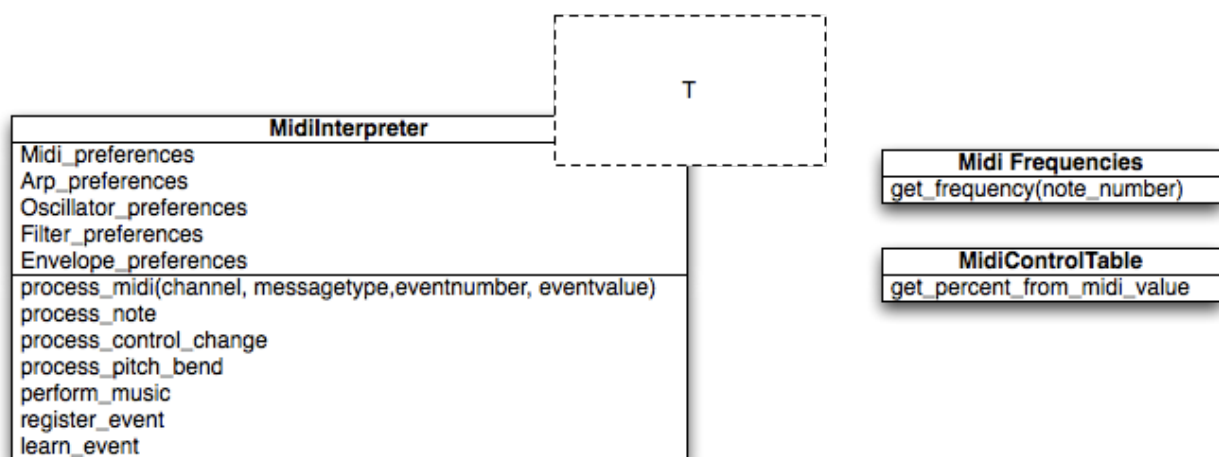


Figure 7 - class diagram for main interpreter

Event flow

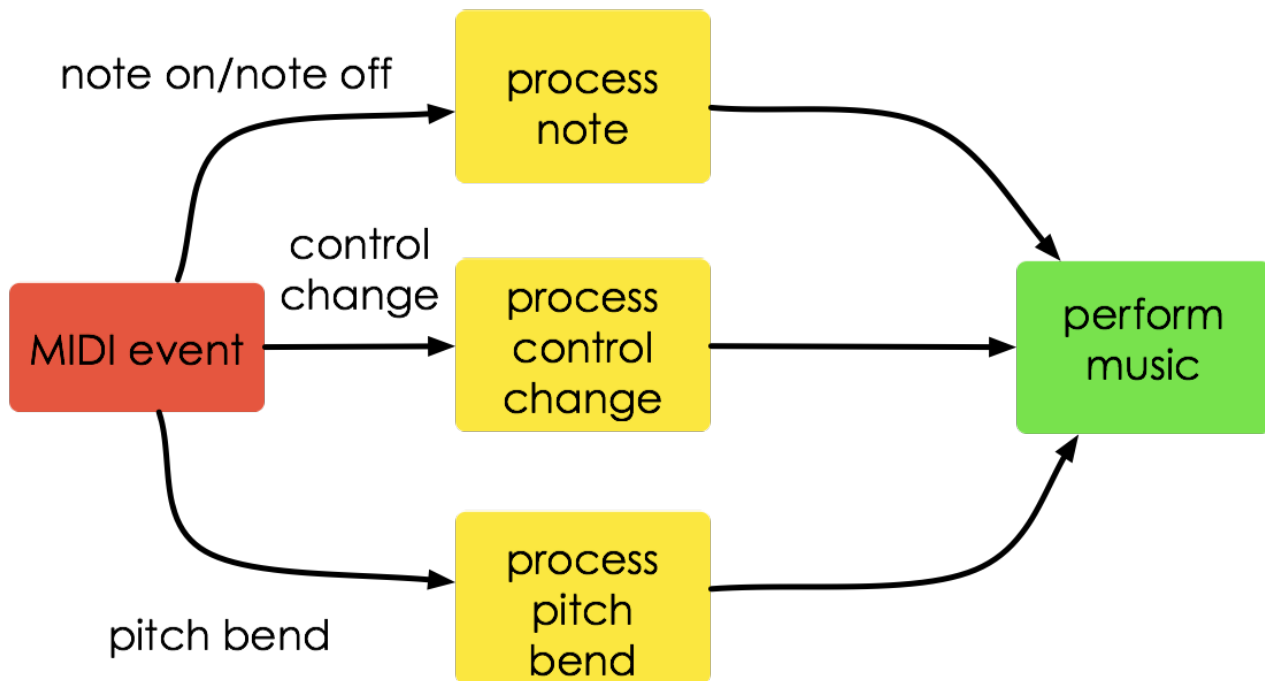


Figure 8 – Flow of MIDI events through the interpreter

Attributes

Midi_preferences, Arp_preferences, Oscillator_preferences, Filter_preferences, Envelope_Preferences

These attributes link to classes that hold a collection of grouped item controllers, and overriding methods for each respective function of the synthesizer. For example, oscillator_preferences contains all the item controllers pertaining to each oscillator frequency offset and volume, as well as the methods and logic needed to get and set their values.

Event_bindings

A dictionary data structure containing all item controllers mapped to particular MIDI control numbers

Methods

Process_midi(channel, eventtype, eventno, eventvalue)

This method is called whenever a full midi packet arrives at input. Its purpose is to route the packet and it's values to a method that will go and perform something functional – eg `process_pitch_bend`, `process_note`, `process_control_change`

Process_note()

This method is called from **process_midi**. Its role is to append or remove the provided note from the list of currently pressed notes, apply a note priority scheme, and call **perform_sound** to make any changes to output audio

Process_control_change(channel, controlnumber, controlvalue)

Called from **process_midi**. This method checks the **event_bindings** table to check whether an item controller has been associated with the supplied controlvalue variable, and runs it's **set_value_midi** if applicable.

Process_pitch_bend()

Called from **process_midi**. Its role is to process input from the pitch bend wheel. If notes are currently active, it alters the current frequency of the synth up or down by the a proportion of up to 2 octaves, depending on the magitude at which the wheel is pushed, and the “bend range” variable item controller. **perform_sound** is then called to make any changes to output audio.

Perform_sound()

Called by **process_pitch_bend**, and **process_note**, as well as a number of item controllers requiring real-time change to output sound on their values changing. Sets all oscillator frequencies (which may have been altered by these functions) by calling the communications subsystem, and triggers the envelope and arpeggiators if they are turned on.

Register_event(item_controller,control_number)

Registers the supplied item controller in the **event_bindings** table, and to the supplied control number, such that said item controller's **set_value_midi** method is called should a midi control event with **control_number** occur.

Learn_event(item_controller)

Places the interpreter into “learning mode”, such that the next midi control event that arrives is automatically registered and mapped to the supplied function . This function is used to learn controls for particular parameters in the user interface.

Timing example – Note-on event

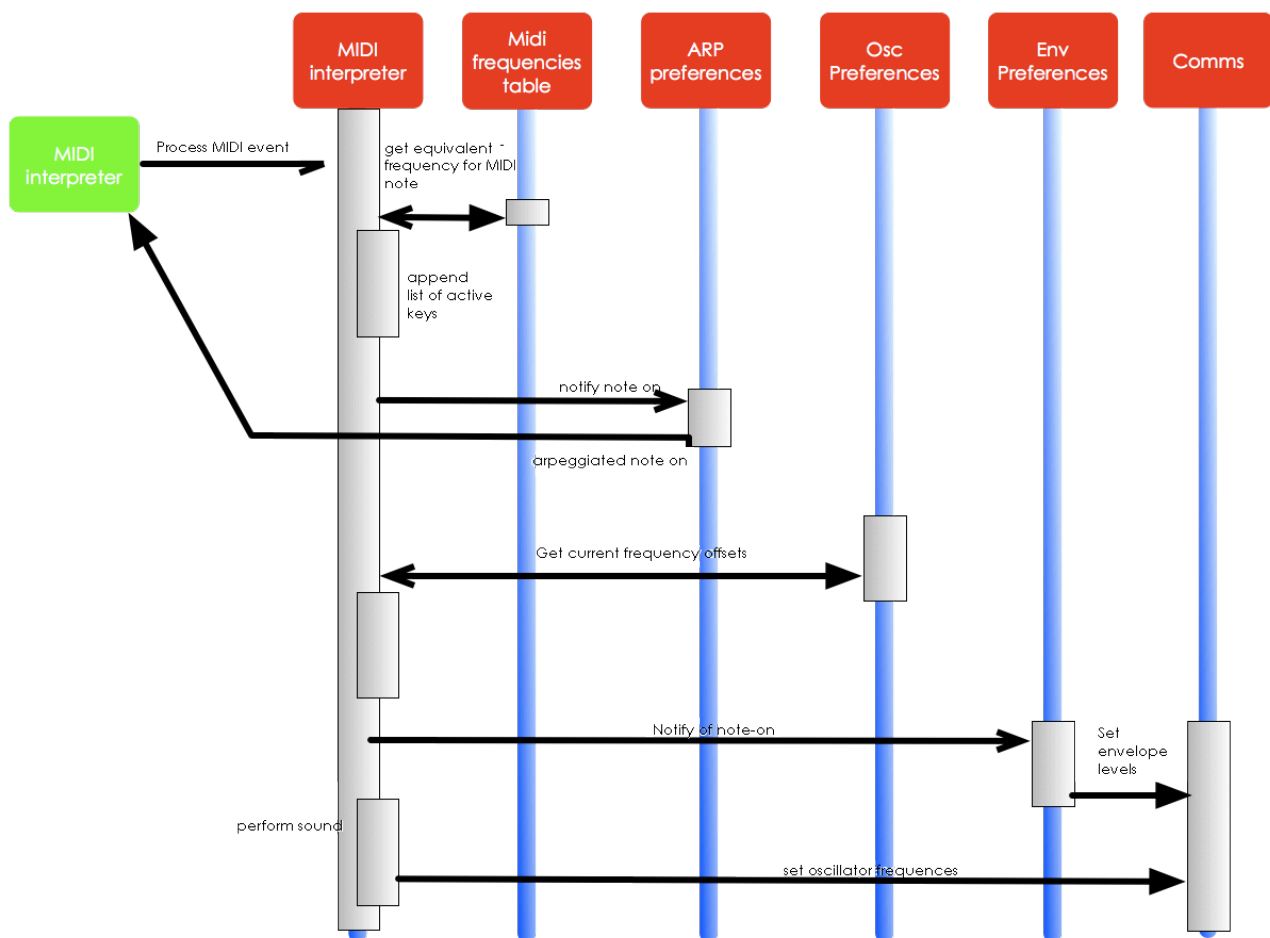


Figure 9 . MIDI Interpreter note-on timing

Figure 9 shows a timing example for a note-on event and how it is processed by the interpreter and it's various interconnected classes. The sequence of events is as follows:

1. A MIDI event arrives at the system and is handled by **process_midi**. Process midi passes this event on to **process_note**.
2. **Process_note** appends the list of currently pressed keys, and calls a note priority scheme in order to convert the list of notes into a particular frequency value.
3. **Process_note** calls **perform_sound**
4. **perform_sound** checks frequency offset amounts from the oscillator item controllers, and applies them to create 3 separate frequency amounts for oscillators 1,2, and 3
5. **perform_sound** checks whether the arpeggiator and envelope generators have been turned on, and sends **trigger** signals to them to perform their given functions
6. **perform_sound** calls the communications layer to set the frequencies of each oscillator to the amounts calculated in step 4 on the FPGA.

Arpeggiator

Technical overview

The arpeggiator's role is to take a note of base frequency, f_b (provided by the MIDI interpreter) and create a sequence of notes based upon it using a particular arp algorithm. This sequence is then iterated through at a user specified rate.

The algorithms implemented by our software arpeggiator are:

- **Up** – takes the base frequency and successively increases it by 1 octave (ie successively multiplies it by 2)
- **Down** – successively decreases the base frequency by 1 octave (ie successively divides it by 2)
- **Up + down** – Performs an up-arp and then “slides” back to base frequency by iterating back through the sequence table in reverse.

Figure 10 illustrates the effect of the various implemented arpeggiator styles upon frequency with respect to time.

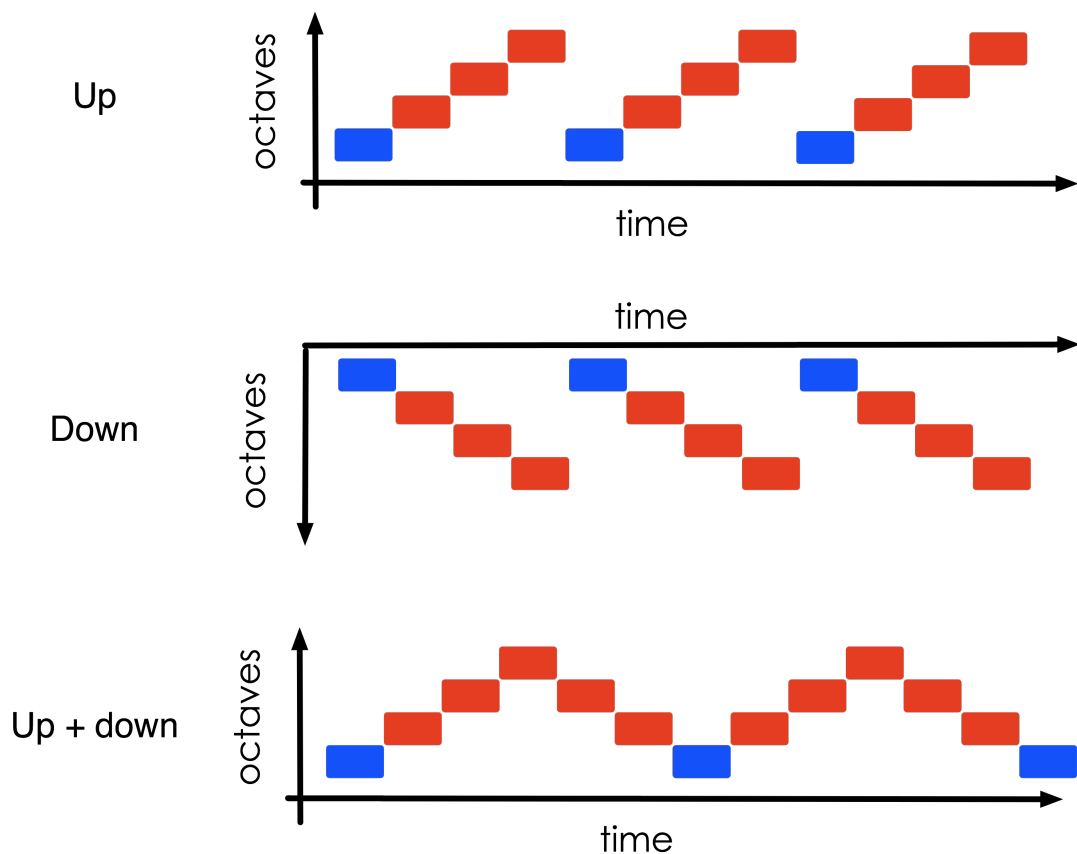


Figure 10 - Arpeggiator styles. The blue rectangles represent the input base frequency, whilst the red represent generated notes

Connection to the MIDI interpreter

The arpeggiator is connected to the MIDI interpreter's **note_on()** method. It's function is triggered and cancelled by this method, and additionally the notes it produces at given time intervals pass through this function. Additionally, arpeggiation parameters such as note-switch rate are dynamically set by item controllers.

Algorithm

The arpeggiator uses an algorithm that creates a sequence when a key is pressed. This sequence is not recalculated until the list of active keys changes. It is iterated-through by a basic timing mechanism upon it's own thread.

Envelope Generator

Technical overview

The envelope generator exists as software upon the control hub, and is used to set the synthesizer's volume over time. It is activated from the **process_music** method of the main MIDI interpreter. The contour used is generated using a graphical “drawing surface”, and volume is set across the following time regions:

Attack-Decay – Activated when a note is initially pressed. This defines the time taken, as well as the set of volume points in the range $0 \leq t \leq t_1$

Sustain – This sample is looped as for as long as a key is held down. Defines the set of volume points in the range $t_1 \leq t \leq t_2$

Release – Iterated when a key is released on the instrument. Defines the time to hold until oscillators are turned off, as well as a set of volume points in the range $t_2 \leq t \leq t_{\max}$

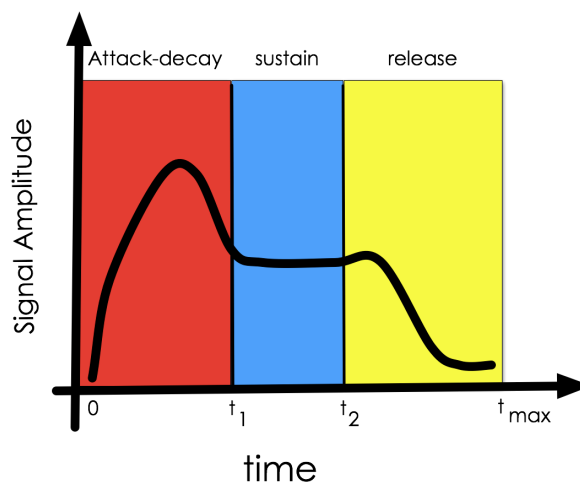


Figure 11. Envelope regions in the time domain

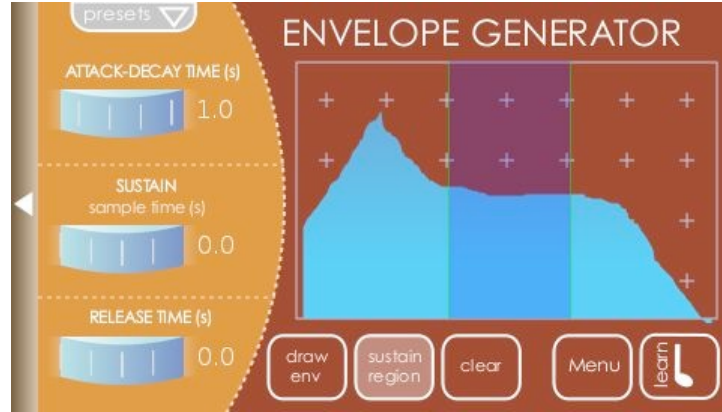


Figure 12. Envelope generate developed as part of the control hub software. The envelope region is overlaid in blue, and attack, sustain and release times are set using the rollers on the left-hand-side of the screen.

Drawing surface

The drawing surface is implemented as a GUI widget. It allows the user to draw-in the envelope and mark-out the sustain region as required.

Drawing creates a set of points, \mathbf{p} , defining the envelope shape.

Sample acquisition

The drawn envelope is sampled using a linear interpolation algorithm to create a set of equally-spaced amplitude points with respect to time.

The number of samples collected for a region is dependent upon the attack-decay time, sustain time, or release time specified by the user.

The set of samples collected for a region (ie. attack, sustain, or release) is defined can be described by the following formulae:

Let x_1 be the left boundary coordinate of the region

Let x_2 be the right boundary coordinate of the region

Let δ be the sampled interval width

Let T be the total time for which the region exists over the region $p(x_1, x_2)$

Let f_s be the sampling frequency

Interval width, $\delta = \frac{x_2 - x_1}{Tf_s}$

Let s be the set of sampled points

Let $f(x)$ be the linearly-interpolated amplitude of the drawn envelope at point x

For each point $n\delta \in \{p(x_1, x_2)\}$:

$$s(n) = f(n\delta)$$

Sample scaling and transmission

The acquired samples must then be scaled to integer values that can be sent to the FPGA

ADC:

Let h be the height (in pixels) of the drawing surface

Let b be the audio bit depth of the system

Let $s_{adjusted}$ be the set of scaled samples, such that

$$s_{adjusted}(n) = \frac{h - s(n)}{h} * (2^b - 1)$$

These samples may now be sent to the FPGA over the communications interface at rate f_s

Graphical User Interface

Technical overview

The GUI has been optimized for use on the gumstix' touchscreen LCD. It consists of a main window, which invokes “panels”, which are collections of widgets that are bound to item controllers. The overall class abstraction is shown in figure 13

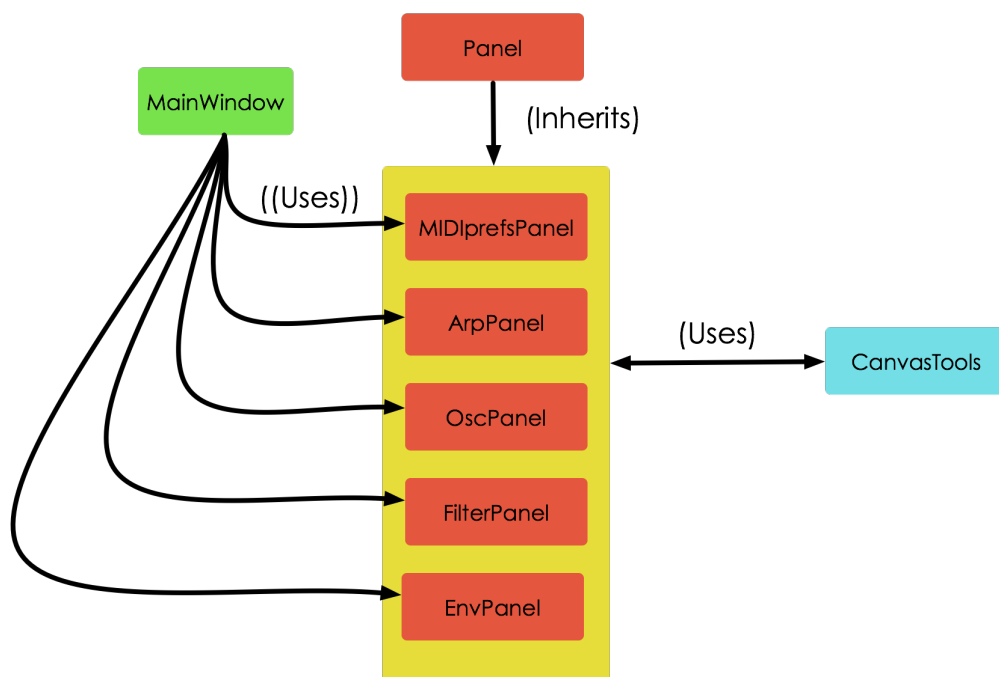


Figure 13. Class hierarchy of the GUI

Development Process

The GUI was prototyped developed in multiple stages:

1. The complete layout of widgets and panels was mocked-up in Adobe Illustrator and assessed for ease of use and adequate functionality

2. Mockup panels were exported to Scalable Vector Graphics (SVG) format
3. Drawing information from the .svg files is extracted using a standard text editor, and converted into goocanvas drawing calls in python classes
4. Widgets were given control and mouse-pointer logic by implementation as goocanvas primitive objects using the Cairo2D graphics library
5. Vector panel backgrounds implemented in python were converted into static PNG images and pre-rendered for optimal speed on the gumstix platform
6. Common panel functionality (eg menu buttons, navigation buttons) was assessed and written into CanvasTools such that it could be reused across all panels
7. Panel navigation, widget binding logic, and transition animations were implemented into the main window

Widgets

Widgets were written based on mock-up images, and intend to provide the best possible user experience in a touchscreen environment. Special consideration was taken to create them such that they could easily be controlled by finger gestures, and to ensure that they had varying levels of sensitivity depending on the task they are connected to.

The first step in creating the set of widgets was to create a goocanvas compatible **abstract** widget, from which all other widgets would subclass, in order to provide common functionality across the GUI system.

Figure 14 illustrates the subclassing of the abstract widget (Item).

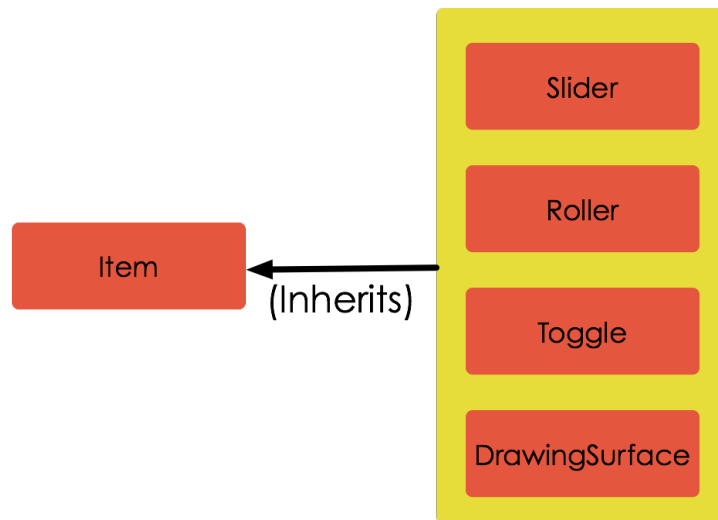


Figure 14. GUI widgets and their inheritance heirarchy

Widgets implemented

Table 1 shows the widgets developed as part of the GUI, and their purpose.




Widget Name	Purpose
Slider 	Easily slides between a range of values on screen. Can have 'snap points' such that a set of discrete points may be selected from
Roller 	Useful for sensitive parameters. Provides a high precision control whose value changes by small increment
Toggle 	Used for on/off parameters

Table 1. GUI widgets and their properties

Common methods

Figure 15 shows the class diagram for all GUI widgets implemented.

Item
signal - learn_me
signal - value_set
do_simple_create_path
do_simple_paint
do_simple_is_item_at
force_redraw
on_motion_notify
on_value_changed
on_learn_me
on_button_press
on_button_release

Figure 15. Attributes and methods of the abstract GUI item from which all of the synths widgets are inherited

do_simple_create_path, do_simple_paint

Called by the panel canvas object. Renders the widget according to its set values

do_simple_is_item_at(x1,y1,x2,y2)

Called by the panel canvas object. Used to check whether the object has been drawn at/is manipulatable using mouse-pointer events at the boundaries specified

force_redraw

Completely re-renders the widget.

on_motion_notify, on_value_changed, on_button_press, on_button_release

Event responders for mouse pointer events. Behavior will vary between each widget implementation. This method usually has the following purposes

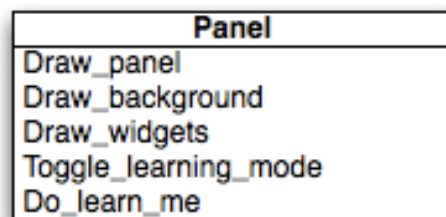
- Redraw the widget according to the event (eg move a slider knob position after a finger slides over it)
- Send a signal to notify a bound item controller that the widgets value has changed

toggle_learn_me

- Places the widget into learning mode (described in section 2), and draws a red transparent overlay to signify this
- Sets a “learning mode” flag, which alters the widget’s behavior such that it is only listening for **button_release** events, which in-turn trigger **learn me** signals to the MIDI subsystem

Panels

In order to conserve memory, panels are drawn on an “as-needed” basis. In other words, they (and their contained) widgets are destroyed when they are not visible on the LCD. Figure xx shows the class abstraction of a typical panel object.



Draw_background(:svg=False)

Paints a panel’s background image using goocanvas. If the **svg** flag is set to true, the background will be rendered live and resolution independently using vector graphics. Otherwise if **svg** is set to false, the panel will be rendered from a static, pre-rendered .png image.

Draw_widgets

Draws and lays-out all contained widgets and returns them as a list such that they can be bound to their corresponding item controllers

Toggle_learning_mode

Toggles learning mode for all widgets contained within the panel

Root canvas/ Main window

The root canvas is responsible for drawing all of the panels onto the synthesizer's display. It also handles animation and navigation, and is responsible for binding the widgets contained in panels to their respective item controllers.

Panels are loaded on an as-seen basis as shown in Figure 16. When a panel is navigated to, the main window adjusts its viewing boundaries (i.e. the part of the coordinate space that is currently visible in the screen), animates, and then destroys any panels that cannot be seen in the these boundaries.

The main panel also intercepts “learn me” signals from any widgets (this is also done by the MIDI interpreter) and uses them to display a “learning” dialog to the user

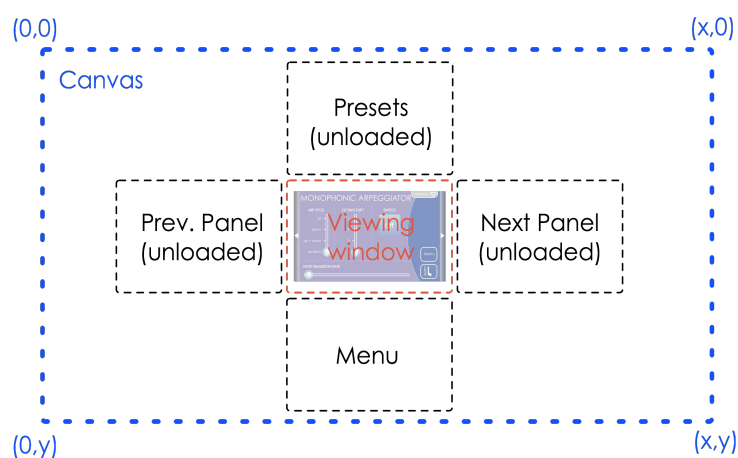


Figure 16. Coordinate system used for rendering panels by the main window

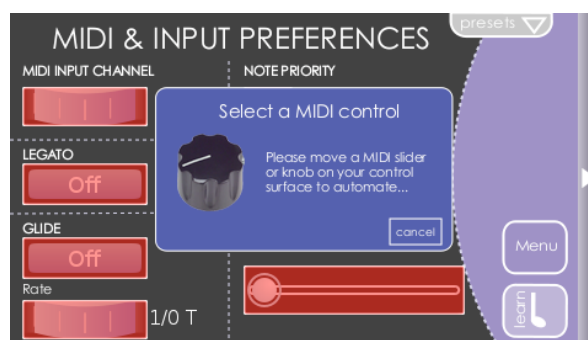


Figure 17. Learning dialog drawn by the main window upon interception of a widget “learn_me” signal

Project case

A project case was constructed for the system using scrap 9mm MDF and pine. It was designed prior to final integration and as such its design requirements were that it would be able to adequately house all components

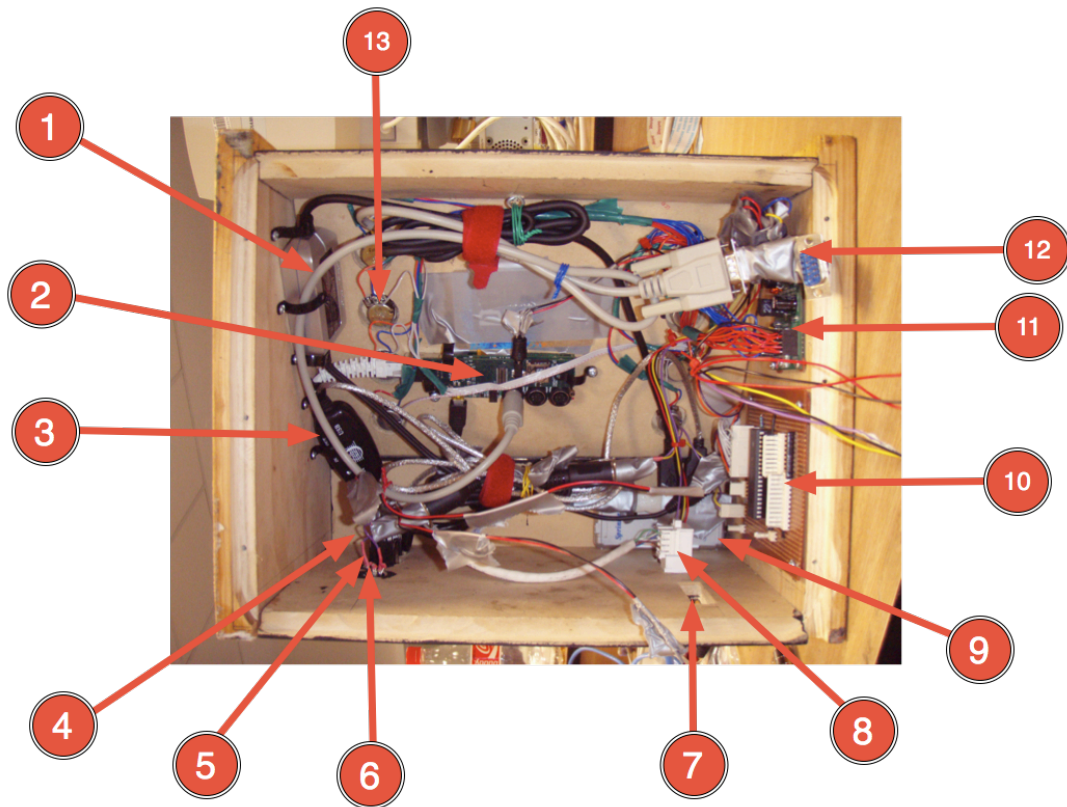


Figure 18. Synth system components (without FPGA and analogue system)

Figure 18 shows the internal layout of the synthesizer without FPGA and analogue systems integrated.

Components shown in Figure 18 are as follows:

1. Roland UM-1 USB MIDI controller
2. Gumstix Control hub
3. Generic USB MIDI controller
4. Power input
5. Line in/out ports
6. MIDI in port
7. Parallel JTAG input (for reprogramming FPGA whilst the synth is still inside it's box)
8. Ethernet I/O
9. USB hub
10. Controller for built in surface
11. 12V-5V rectifier & down converter for control surface and FPGA
12. Serial communications link to FPGA
13. Built-in surface pots (x8)

The analogue system and FPGA were screwed to the floor of the case:

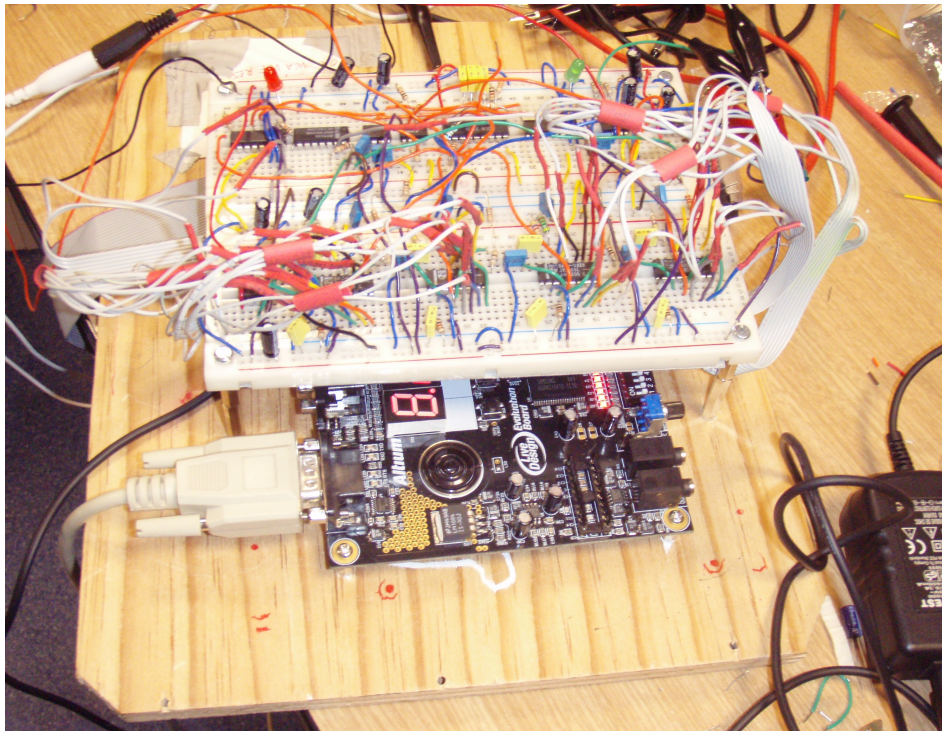


Figure 19. FPGA and analogue implementations mounted on project case bottom

Time management

Milestones

A number of milestones were set by the group whilst undertaking the project. These are as follows:

1. Understanding of Synthesis elements

Commencement Date: Week 6, Semester 1

Completion Date: End of Week 12, Semester 1

Major Outcomes:

1. Developed an understanding of basic synthesis, existing synthesizer functionality, and the ways it can be controlled
2. Performed matlab simulations to ensure knowledge of signal processing
3. Ensured adequate Altium designer knowledge
4. Assessed and purchased required equipment to develop synthesizer

Completed by:

- 1- Ben Davey, Nathan Sweetman, Aaron Frishling, Andreas Wulf
- 2- Ben Davey
- 3- Nathan Sweetman, Aaron Frishling
- 4- Ben Davey, Aaron Frishling, Andreas Wulf, Nathan Sweetman

Details:

This milestone essentially involved gaining an understanding of the task ahead of us. We performed matlab simulations to ensure our basic understanding of the fundamental machinations of existing synthesizers. This then helped us to assess the essential functions of our synth, as well as assess how we would implement the device in the allocated timeframe and with what components

2. Completed setup of development environment

Commencement Date: Week 12, Semester 1

Completion Date: Week 5, Semester 2

Major Outcomes:

1. Assessed and deployed required 3rd party software packages onto Gumstix
2. Compiled custom gumstix kernel
3. Installed all required packages on host development machines (python, bitbake, etc)

Completed by:

- 1-Ben Davey, Nathan Sweetman
- 2- Andreas Wulf, Ben Davey

Details:

This milestone became an ongoing one as we constantly reassessed what software was required for the system as other implementations failed or had poor functionality on the gumstix platform. Kernel modules were also recompiled at various points in time to fulfil requirements. At times this often took much longer than we anticipated due to cross-compilation errors and dependency troubles. We found that our initial MIDI subsystem (which worked on our host development machines) did not work on the gumstix due to ALSA's apparent lack of support for USB sequencer devices. Thus it had to be rewritten to function with available resources.

3. Completed Development of core MIDI and oscillators

Commencement Date: Week 1, Mid-year break

Completion Date: Week 3, Semester 2

Major Outcomes:

1. Implemented basic MIDI interpreter on the gumstix platform
2. Developed basic oscillator functionality with a single waveform

Completed by:

- 1- Ben Davey
- 2- Aaron Frishling, Nathan Sweetman

Details:

This milestone entailed the development work of both the MIDI interpreter and oscillator sub-systems in unison, such that they could be integrated. Basic oscillator functionality was provided relatively early in the project timeline, however it had to be reworked a number of times due to a the lack of processing capability on the FPGA and our lack of foresight in the development of an oscillator system providing multiple waveforms.

4. Completed GUI mockup

Commencement Date: Week 12, Semester 1

Completion Date: Week 8, Semester 2

Major Outcomes:

1. Developed a mockup user interface using Adobe Illustrator
2. Developed all GUI widget components and installed required library packages onto the gumstix
3. Converted UI mockups into working code

Completed by:

1 - 4. Ben Davey, Andreas Wulf

Details:

Development of the GUI was a multi-stage task - We had initially envisaged developing it entirely using Python/PyGTK+, however we discovered that it did not provide the necessary functionality required to make the interface we desired. After several weeks of re-assessment, we decided to make use of the GooCanvas rendering library, however this was also fraught with initial problems - chiefly the lack of documentation provided with the library, and difficulties getting it compiled to work on the gumstix platform. This milestone was finally achieved to our standards after several arduous months of work and implementation of non-existent functionality from scratch.

5. Control systems developed

Commencement Date: Week 6, Semester 2

Completion Date: End of Week 6, Semester 2

Major Outcomes:

1. Developed communications protocol between gumstix and FPGA for synth controller
2. Implemented protocol on gumstix and FPGA systems
3. Developed gumstix MIDI interpreter and link to communications protocol such that hitting a note produces an audible tone on the oscillators

Completed by:

1&2 - Nathan Sweetman

3- Ben Davey, Nathan Sweetman

Details:

Perhaps the most important milestone in terms of the synthesizer's functionality and group morale; this was the point where several components that had been developed in unison were finally integrated, and was the point when the system became a functioning **musical** synthesizer, rather than merely a set of discrete systems.

6. Completed analogue systems

Commencement Date: Week 12, Semester 1

Completion Date: Week 2, Semester 2 break

Major Outcomes:

1. Integrated CEM3889
2. Developed vocoder and post-filter modulation effects
3. Interfaced analogue systems with FPGA
4. Provided control system for analogue systems

Completed by:

1 - 4. Nathan Sweetman, Aaron Frishling

Details:

The analogue systems were developed over the entire project timeline. Initially we designed our own voltage controlled filter, and deciding that it did not meet our requirements, we decided to purchase a system that did on an IC - the CEM3889. This component was tested using a function generator over the midyear break, but did not see complete integration with

the rest of the system until the end of the semester 2 holidays. Additionally, vocoder and digital control was implemented at the end of the project timeframe (although the protocol had been devised earlier, and code was ready for controlling these systems upon the FPGA and gumstix much earlier)

7. Implementation of arpeggiator and envelope generator completed

Commencement Date: Week 6, Semester 2

Completion Date: End of break, Semester 2

Major Outcomes:

1. Develop basic arpeggiator system
2. Develop Envelope generator

Completed by:

1 & 2. Ben Davey

Details:

The arpeggiator was not an initial requirement of our system, however it was decided half-way through the project's timeline that it would be a useful feature and would aid in showcasing the system. Development of the arpeggiator was a relatively straightforward process (completed in 1 day), however getting it controlled correctly through MIDI took much longer, and much debugging and thread hacking was necessary.

8. Integration of control systems and GUI

Commencement Date: Week 1, Semester 2 break

Completion Date: End of Week 2, Semester 2 break

Major Outcomes:

1. Integrate MIDI subsystem with GUI
2. Refine GUI and widgets to meet control system requirements

Completed by:

1 & 2. Ben Davey, Andreas Wulf

Details:

This milestone represented our last major accomplishment towards getting the synthesizer functionally complete. It involved developing a project case, integrating all components inside it, making the built-in surface functional, and developing the GUI item controllers. This took the duration of the break in semester 2.

9. User acceptance testing and general refinement

Commencement Date: Week 9, Semester 2

Completion Date: End of Week 10, Semester 2

Major Outcomes:

1. Assess bugs and errors
2. Develop bug-fixes and workarounds
3. Refine GUI and user experience
4. Optimize GUI components for gumstix
5. Optimize FPGA oscillators and analogue systems for best audio quality

Completed by:

- 1 & 2. Ben Davey, Andreas Wulf, Nathan Sweetman, Aaron Frishling
3 & 4 . Andreas Wulf
5. Nathan Sweetman, Aaron Frishling

Details:

2- Aaron Frishling, Nathan Sweetman

10.Integration into case

Commencement Date: Week 8, Semester 2

Completion Date: End of Week 2, Semester 2 break

Major Outcomes:

1. Create project case
2. Integrate all systems

3. Complete project

Completed by:

1. Ben Davey

2 & 3. Ben Davey, Aaron Frishling, Andreas Wulf, Nathan Sweetman

Details:

This milestone required putting the completed synthesizer components through their paces and resolving any unnoticed bugs contained.

Conclusion

The completion of the synthesizer marked a great personal achievement for all members of the group. It was the biggest project any of us have worked upon to date, and by far the most rewarding.

Developing the synthesizer involved applied knowledge of Analogue systems, digital electronics, embedded development, data structures, operating systems, and event-driven paradigms. It's large scope further enhanced our ability to manage time, and to take an abstract view of an inherently complicated system by breaking it into small parts.

The project was not without it's frustrations. Several elements took much longer to develop than was initially anticipated, and on numerous occasions we had to apply our resourcefulness to develop workaround solutions for problems that seemed as though they were going to take forever to complete continuing along our initial design plans.

In hindsight and since completion, there are a number of fundamental design changes our group would make in future iterations of the synthesizer. Perhaps the most notable of these is the design of our oscillators on the FPGA; the use of a softcore processor whose maximum clock speed is only just capable of achieving our oscillator and mixing requirements meant that any code written needed to be heavily optimized, and leaves little leeway for the implementation future features on this component. Developing the oscillators as using a custom VHDL processing solution with DMA features may have overcome this, however this would have taken far more time and resources than we were allocated. Also given that FPGAs are a rapidly developing technology, the problem we faced is unlikely to continue into the future.

As synthesizers are a technology with near unlimited potential for change and layering of additional elements, the room for improving the system with more features and effects is almost limitless. Perhaps the most important feature we would add to the system would be an additional analogue filter whose cutoff frequency and resonance is controlled by a key-driven envelope. This is a feature of several other synthesizers, would be relatively simple to implement. Its effect would greatly increase the range of sounds the synthesizer is capable of producing.

The vocoder is also a feature that could be greatly improved in future revisions. Passing input through a multi-band filter system where the output from each filter is modulated by a carrier of different frequency (offset by the keyboard) before it is finally additively mixed again may lead to much more desirable audio output, particularly when vocoding human voices.

Our goals were to deliver a musical synthesizer with a sound that is audibly appealing, and rich in features and customization options whilst being simple to use and whose learning curve was well within the reach of the non-technically minded. We wanted a synthesizer that was not reliant upon additional hardware and software, and that could easily be played in a live environment. We delivered all of these goals, and we hope that our creation is enjoyed as much by its users in playing it as it was for us developing it.

References

- Gumstix company website : <http://www.gumstix.com>
- Pango text rendering library : <http://www.pango.org/>
- Cairo 2D graphics rendering library : <http://www.cairographics.org/>
- GooCanvas documentation : <http://library.gnome.org/devel/goocanvas/unstable/GooCanvas.html>
- PyGTK - GTK+ bindings for python : <http://www.pygtk.org/>
- Gobject reference manual : <http://library.gnome.org/devel/gobject/unstable/>
- Wikipedia – Musical Instrument Digital Interface : <http://en.wikipedia.org/wiki/MIDI>
- Wikipedia - Microwindows embedded graphics environment: <http://en.wikipedia.org/wiki/Microwindows>