UNIVERSITY OF THE
WEST OF ENGLAND

Audio Signal Processing on the Parallax Propeller Multiprocessor Microcontroller

Peter Hemery
08014298

2011/2012

Supervised by Dr. Rob Williams

Word Count: 12,124

Abstract

Multi-core parallel processing has become a popular feature in modern CPU design. This paper investigates the suitability of the Parallax Propeller as an educational platform for introducing Digital Signal Processing techniques on a multi-core architecture in an academic environment.

The Propeller P8X32A contains eight identical 32-bit microprocessors and supports Object Orientated design. Propeller Objects are distributed via the Propeller Tool’s software library and the Propeller Object Exchange website. They are used as building blocks to build applications by encapsulating functions and variables within a single file. This enables the developer to reuse prebuilt library and driver code for application specific functionality, requiring only knowledge of the interface and not the code behind it. Objects may include any number of other objects and each object may dynamically utilise any number of available processors to perform the task required, such as hardware control, peripheral communication or data processing.

The target platform will be the Parallax Propeller Demo Board. It features PS/2 Keyboard and Mouse input ports, VGA and composite graphical output, USB communication, built-in microphone, headphone jack, eight General Purpose I/O pins and a breadboard for peripheral devices.

Several DSP applications that exploit the multi processor environment have been ported for demonstration. Particular focus has been given to Audio Signal Processing for investigating the capabilities and limitations of the Propeller on the Parallax Demo Board for audio analysis and graphical representation. Frequency analysis techniques are presented together with obtained performance results.

Table of Contents

Table of Figures

1 Historical Context and Motivation

During the second half of my second year of University in January 2010, I went on foreign exchange to the University of Central Florida (UCF). I enrolled on a class taught by Dr. Alfred Ducharme, the head of the Computer Engineering department. His class turned out to be about the basics of computing, which I had already studied in my first year, so I approached him and asked if there was anything I could be involved in that was more challenging. He introduced me to two seniors who were deciding on their Senior Design project.

Dr. Ducharme proposed the use of a robot called the Stingray, which uses the Parallax Propeller microcontroller, to implement a self-navigating, obstacle avoiding solution using GPS, compass and ultrasonic sensors. The Propeller is the result of 8 years hard work by former UCF student Chip Gracey. He designed the 8-core microcontroller and two programming languages to run on it, Spin (a high-level, object-based language) and Propeller Assembler (PASM). To get me up to speed with using the Propeller I was provided with a Demo Board with a manual. My understanding progressed throughout the course of the semester and by the time I had finished I was pleased with the results I had achieved with the Stingray. Before I left Florida Dr. Ducharme told me to keep the Demo Board, since it had been given to him by Parallax to promote educational projects.

During the discussions with my course leader about potential project ideas, it was suggested that the use of the Propeller would be appropriate, so I began to explore different uses for the Demo Board. Searching through the provided code demos and the Parallax Object Exchange (ObEx) site, I discovered a few useful demo applications, including Microphone to VGA. This produced a waveform on the screen taken from live microphone input.

After my return to England I came into possession of a thumb piano, called a Kalimba or Mbira. This musical instrument is very popular in Africa and South America. It consists of 17 tines, narrow and thin curved pieces of metal of varying length, mounted on a hollow wooden block. The tines are usually arranged with the longest tines in the centre with progressively shorter tines towards the edge, with note ascension alternating on each side. There are a few variations in the tuning of this instrument, and the tuning method is a little hap hazardous; using pliers.

The standard tuning for a 17-tine treble Kalimba is below. I realised that the tuning of my Kalimba was wrong as soon as I got it, so I set about trying to tune it using online sound detection applications. This proved moderately successful, although the Kalimba regularly went out of tune after a few hours use.

I decided to set myself a challenge: To use the Propeller Demo Board as a platform to build my own tuner and investigate the possibilities for interactive graphical feedback.

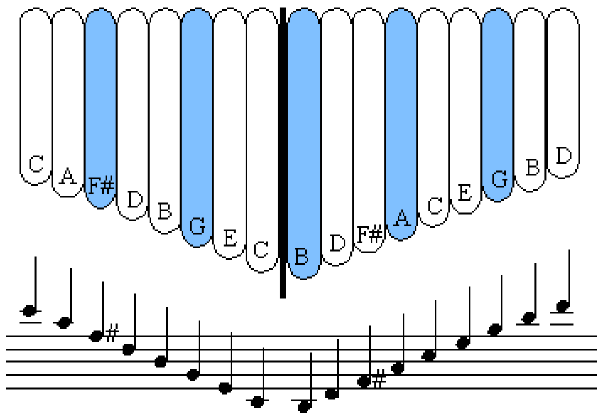


Figure 1 – Kalimba Note Layout [IX] **(Holdaway, 2006-2012)**

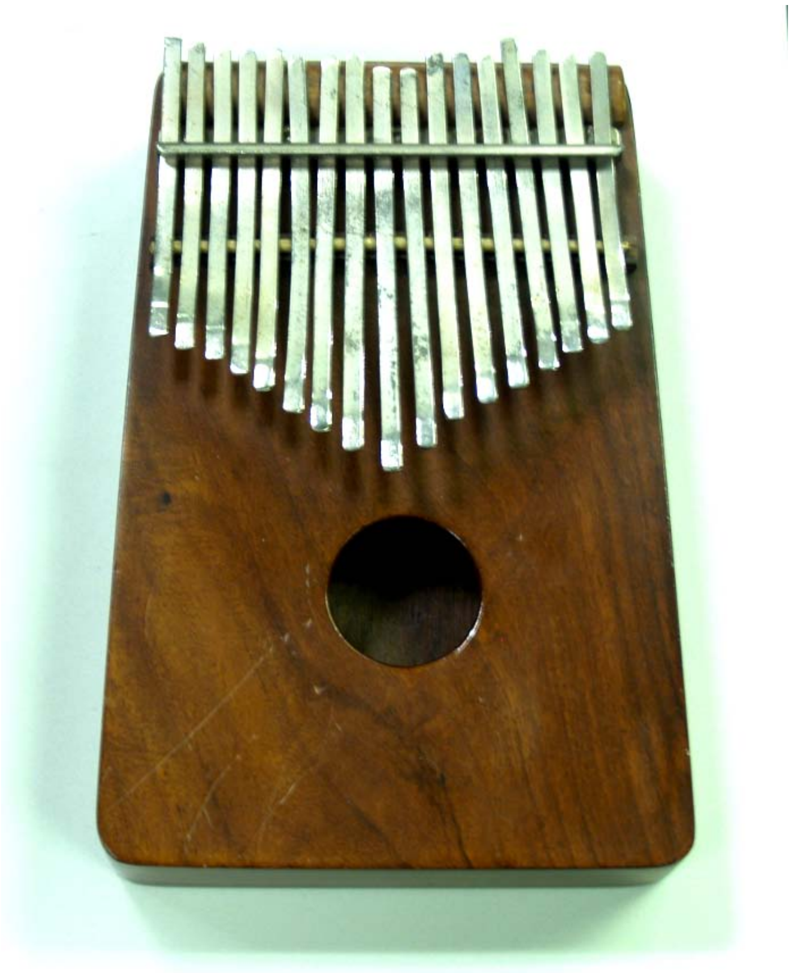


Figure 2 – The Kalimba

2 Project Aims for Audio Analysis

The main goal for this project is to explore and discover the complexities of Digital Signal Processing for audio signals and to gain an understanding of how music can be processed, analysed and represented digitally. For this purpose, the Propeller is a good learning platform.

Many software engineers that have background knowledge of mathematics will be familiar with Fourier but may be unable to code a Fast Fourier Transform. When there is a need for the FFT for a new platform where no existing implementations exist, reverse engineering a C version without knowing how it works becomes an unattractive option. Curiosity about how algorithms work, especially those as remarkable as the FFT, has been a driving force.

2.1 Musical Note Detection

The focus of the project is to detect a musical note produced by an instrument. Two avenues will be explored. First is to determine how close or far a played note is from its ‘true’ form, for the purpose of tuning an instrument. The second is to determine multiple notes played at the same time for the purpose of live score notation. This is a much harder task and will be discussed later.

2.2 Musical Instrument Graphical Tuner

There are many existing solutions for the problem of tuning musical instruments. I already own a basic musical tuner that displays only the notes relevant for guitars. More generic hardware tuners exist and a wide variety of free, proprietary and open source software tuners are available online.

Few of the currently available solutions represent the required information in a clear and eye-catching manner. These are two of the goals for my implementation of the tuner.

2.3 Spectrogram Generation

One of the features of many music analysis programs and devices is the graphical representation of the frequency spectrum in the form of a spectrogram. A spectrogram forms an image that represents how the intensities of component frequencies of a given signal vary over time. These programs and devices are called spectrographs. This feature shall be investigated.

2.4 Musical Score Notation

There are already many different implementations of musical instrument tuners available; however, there are few existing solutions for the live representation of multiple musical notes on a standard musical score. The process of transcribing live music into standard written notation has great practical implications for musicians. This may turn out to be the most difficult aim for the project and may turn into an ongoing process to achieve satisfactory implementation.

2.5 Requirements for Audio Analysis

This section aims to outline the requirements for this project.

2.5.1 User Interface

1. The interface must be informative.
2. The interface should be eye-catching.

2.5.2 Accuracy & Precision

1. The system must be able to detect and discern musical notes.
2. The notes detected should be between 220 Hz and 1500 Hz (A3 to F#6).

2.5.3 Single Note representation

1. The note shall be accurately discernible with at least 20 divisions between each neighbouring note.
2. The note should be accurately discernible with at most 100 divisions between each neighbouring note.

2.5.4 Multiple Note representation

1. The system should be able to detect multiple notes simultaneously.
2. The system should accurately display notes changing over time.
3. The displayed notes may be superimposed onto a musical score.
4. The displayed notes may be superimposed onto a graphical piano keyboard.

2.6 Approach to Development Lifecycle

Development shall be undertaken using incremental throwaway prototypes. Each stage of development will conclude as prototypes achieve required functionality.

3 Introduction to Parallax Propeller Microcontroller

The Propeller architecture is quite unusual. One of the main features is the Multi-Core design, which allows the developer to dedicate a core to process an individual task. For example, one core may be assigned to deal with high bandwidth asynchronous RS232 communication while another core uses its timing hardware to implement various Pulse-Width Modulation (PWM) signals.

Developer driven dynamic cog assignment brings compute flexibility and multi-core run-time management easily solves event-handling problems, eliminating the need for interrupts and providing deterministic timing.

Another main feature advantage of the Propeller is that the code is object orientated. This allows pre-written objects that support many types of hardware to be integrated into a developer’s project with very little effort, simplifying the coding process.

The Propeller is relatively inexpensive. A single P8X32A-Q44 is currently priced at $7.99 on the Parallax website, with the Demo Board priced at $59.99.

This section aims to cover the essentials for understanding the Propeller’s capabilities. The concept of the architecture has been explained quite succinctly in the manual:

> The Propeller chip is designed to provide high-speed processing for embedded systems while maintaining low current consumption and a small physical footprint. In addition to being fast, the Propeller provides flexibility and power through its eight processors, called cogs, that can perform simultaneous independent or cooperative tasks, all while maintaining a relatively simple architecture that is easy to learn and utilize.
>
> The resulting design of the Propeller frees application developers from common complexities of embedded systems programming. For example:

- The memory map is flat. There is no need for paging schemes with blocks of code, data or variables. This is a big time-saver during application development.
- Asynchronous events are easier to handle than they are with devices that use interrupts. The Propeller has no need for interrupts; just assign some cogs to individual, high-bandwidth tasks and keep other cogs free and unencumbered. The result is a more responsive application that is easier to maintain.
- The Propeller Assembly language features conditional execution and optional result writing for each individual instruction. This makes critical, multi-decision blocks of code more consistently timed; event handlers are less prone to jitter and developers spend less time padding, or squeezing, cycles here and there.

[XIV] **(Parallax Inc., 2006-2011)**

The Propeller is particularly useful for applications that can be simplified by concurrent processing. True parallel processing with multi-purpose design increases system capability and resolves the common problem for embedded devices of incompatible bandwidth requirements.

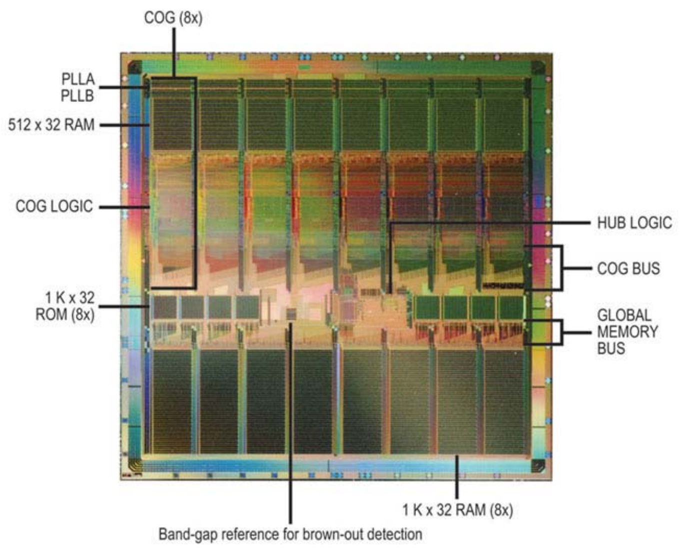


Figure 3 – Propeller Die Diagram [XVI] **(Parallax Inc., 2007)**

Potential applications include: industrial control systems, portable handheld devices, motor and servo control, low-cost video games, synthesisers, emulators, sensor integration, data acquisition and digital signal processing.

The winner of the 2009/2010 Propeller Contest was an Internet Radio Player with MP3 Recording and Playback capabilities, designed by Harrison Pham. Third place was a Spin compiler called Sphinx that runs on the Propeller chip, designed by Michael Park. It can compile complex and substantial programs (including those containing Propeller assembly language) such as the Parallax TV and VGA graphics objects and even Sphinx itself. [XVII] **(Parallax Inc., 2010)**

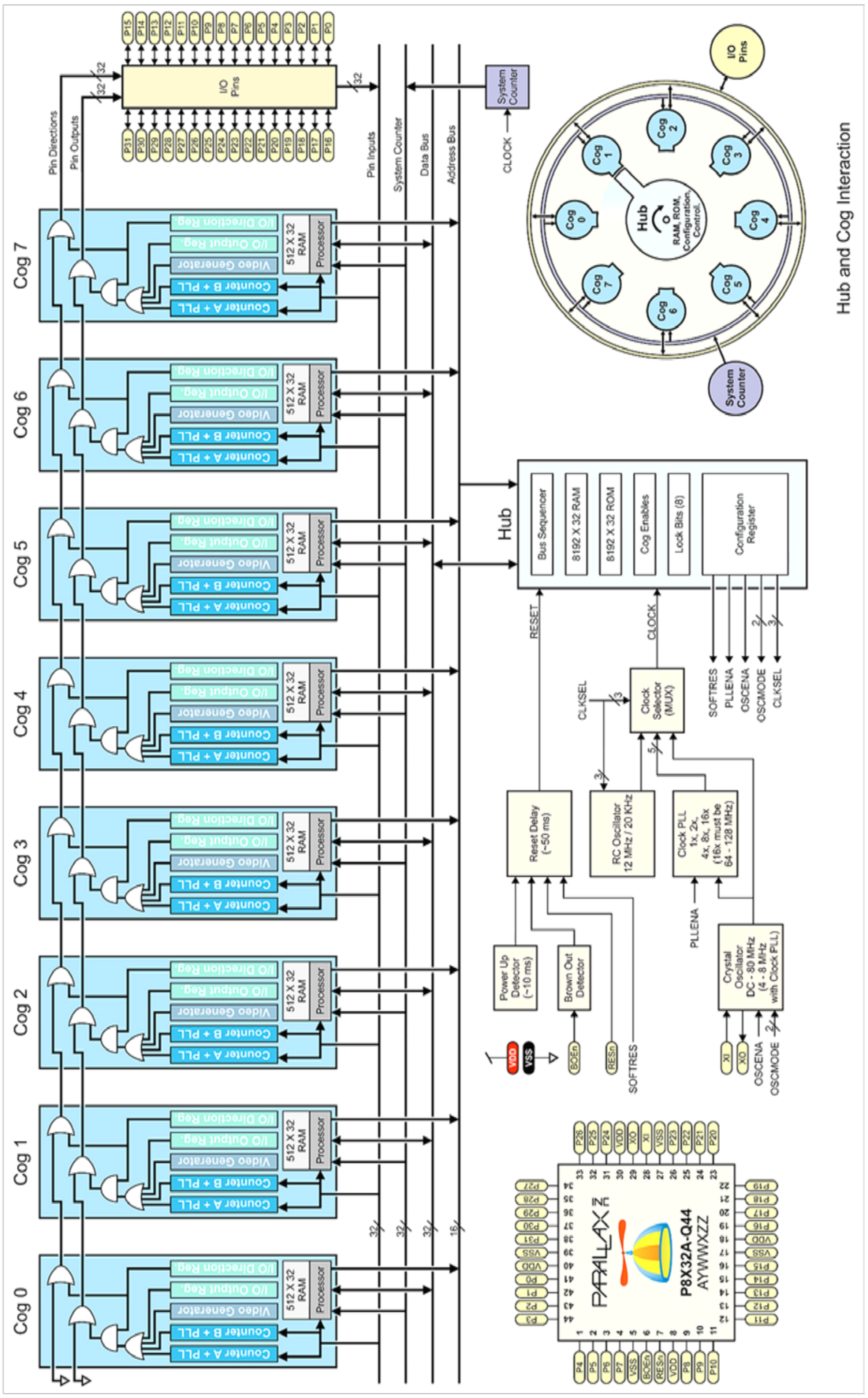


Figure 4 – Propeller Architecture Diagram [XVI] **(Parallax Inc., 2007)**

3.1 Architecture Specification

The Propeller architecture consists of 8 independent 32 bit processors, called cogs labelled 0-7, that each contain: 2KB (512 x 32bit cells) of RAM; two counters with PPLs 1x-16x, a Video Generator capable of PAL, NTSC and VGA; an I/O Output Register and I/O Direction Register. Each cog has mutually exclusive access to shared resources, using an internal bus controller called the Hub, with a Round Robin scheduling scheme. The Hub consists of 32KB of RAM, for program code and variable data, and 32KB of ROM.

The Propeller has two types of shared resources, mutually exclusive and common. Common resources can be accessed at any time by any cog. These include I/O pins and the System Counter. All other resources are mutually exclusive and handled by the Hub.

Data can be defined as byte, word or long (8, 16 or 32 bit) aligned and is stored in little-endian format. This is so that as memory is read a byte or a word at a time it maintains the correct order, even though memory is made up of 32 bit cells. Propeller Assembler allows data to be defined as byte aligned but long sized.

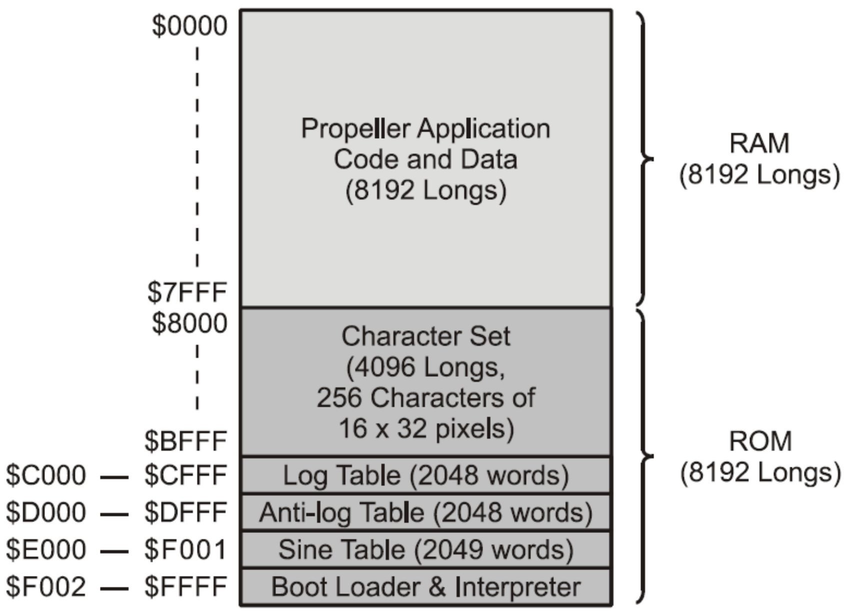

$0000
$7FFF
Propeller Application Code and Data (8192 Longs)
RAM (8192 Longs)
$8000
$BFFF
Character Set (4096 Longs, 256 Characters of 16 x 32 pixels)
$C000 — $CFFF
Log Table (2048 words)
$D000 — $DFFF
Anti-log Table (2048 words)
$E000 — $F001
Sine Table (2049 words)
$F002 — $FFFF
Boot Loader & Interpreter
ROM (8192 Longs)

Figure 5 – Propeller Main Memory Map [XIX] **(Parallax Inc., 2011)**

3.1.1 ROM Contents

3.1.1.1 Character Set

ROM consists of 8KB for the Propeller Font Character Set of 256 Characters of 16x32 pixels each, Word (16 bit) aligned Log, Anti-Log and Sine Tables of 4KB each and 4KB containing code for the Boot loader and its own high-level language interpreter.

The Propeller Font is based on a North America / Western Europe layout and includes special characters for drawing waveforms, schematics, arrows, bullets and Greek characters common in electronic and mathematical notation.

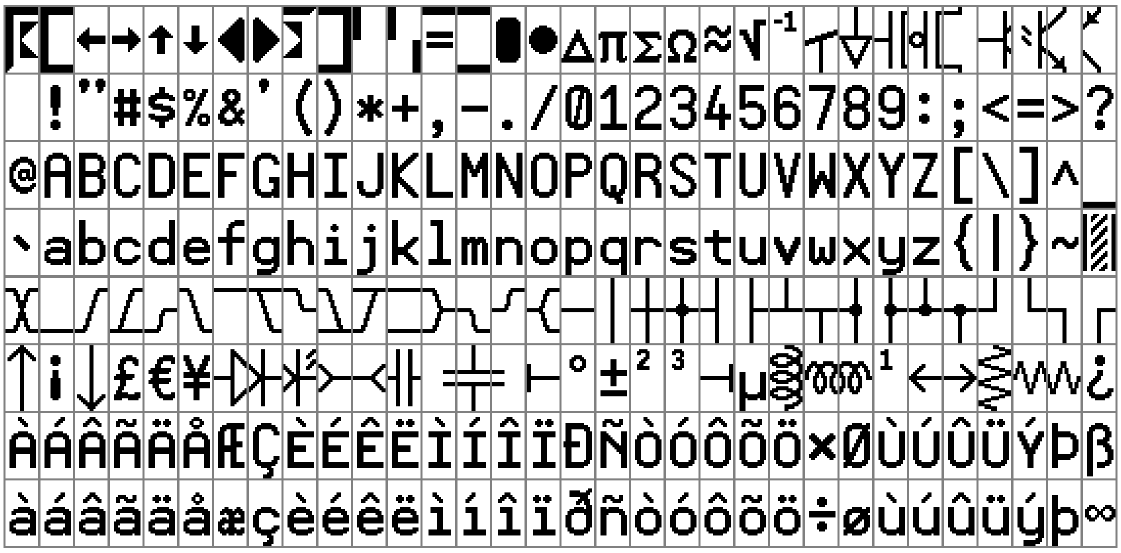


Figure 6 – Propeller Font Character Set [XIX] **(Parallax Inc., 2011)**

3.1.1.2 Log and Anti-Log Table

The log and anti-log tables are useful for converting values between their number form and exponent form. When numbers are encoded into exponent form, simple maths operations take on more complex effects. For example ‘add’ and ‘subtract’ become ‘multiply’ and ‘divide,’ ‘shift-left’ becomes ‘square’ and ‘shift-right’ becomes ‘square-root,’ and ‘divide by 3’ will produce ‘cube root.’ Once the exponent is converted back to a number, the result will be apparent. This process is imperfect, but quite fast.

For applications where many multiplies and divides must be performed in the absence of many additions and subtractions, exponential encoding can greatly speed things up. Exponential encoding is also useful for compressing numbers into fewer bits – sacrificing resolution at higher magnitude. In many applications, such as audio synthesis, the nature of signals is logarithmic in both frequency and magnitude. Processing such data in exponent form is quite natural and efficient, as it lends a ‘linear’ simplicity to what is actually logarithmic. [XVIII] **(Parallax Inc., 2011)**

3.1.1.3 Sine Table

The sine table resides in locations $E000 - $F001 in Main Memory. It can be used for calculations related to angular phenomena. There is also a Log Table and an Anti-Log Table .

The sine table provides 2,049 unsigned 16-bit sine samples spanning from 0° to 90°, inclusively (0.0439° resolution). Sine values for all other quadrants covering > 90° to < 360° can be calculated from simple transformations on this single-quadrant sine table.

A small amount of assembly code can mirror and flip the sine table samples to create a full-cycle sine/cosine lookup routing which has a 13-bit angle resolution and a 17-bit sample resolution. As with the log and anti-log tables, linear interpolation could be applied to the sine table to achieve higher resolution.

[XVIII] **(Parallax Inc., 2011)**

3.1.1.4 Boot Procedure

After boot up all cogs have access to all 32 I/O pins. It is up to the developer to ensure no two cogs use the same pins for different purposes. Pins 30 and 31 are reserved for serial communications with a host PC, while pins 28 and 29 interface to external EEPROM. At boot time the boot loader detects PC host communication and, if detected, identifies the Propeller chip and possibly downloads a program into RAM or optionally the 32KB of EEPROM. If no host is detected the boot loader loads all 32KB of EEPROM data into RAM. If no EEPROM is detected the boot loader stops, cog 0 is terminated and the Propeller goes into shutdown mode, with all I/O pins set to input. If global RAM was successfully loaded then cog 0 is reloaded with the built-in Spin Interpreter and the user code is run.

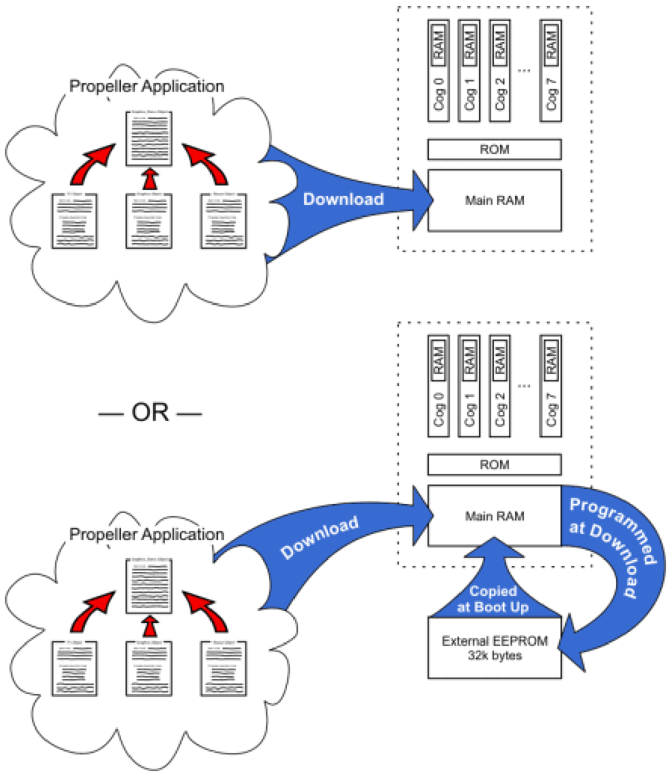


Figure 7 – Downloading Propeller Application

[XVIII] **(Parallax Inc., 2011)**

When a program is loaded, either from a host or external EEPROM the entire Hub memory space is written. The first 16 locations ($0000 - $000F) hold initialization data used by the Boot loader and Interpreter. Developers' executable and data code will begin at $0010 and will extend some way into memory. The area in Hub RAM after executable data is used as variable and stack space. This extends to $7FFF, where Hub ROM begins.

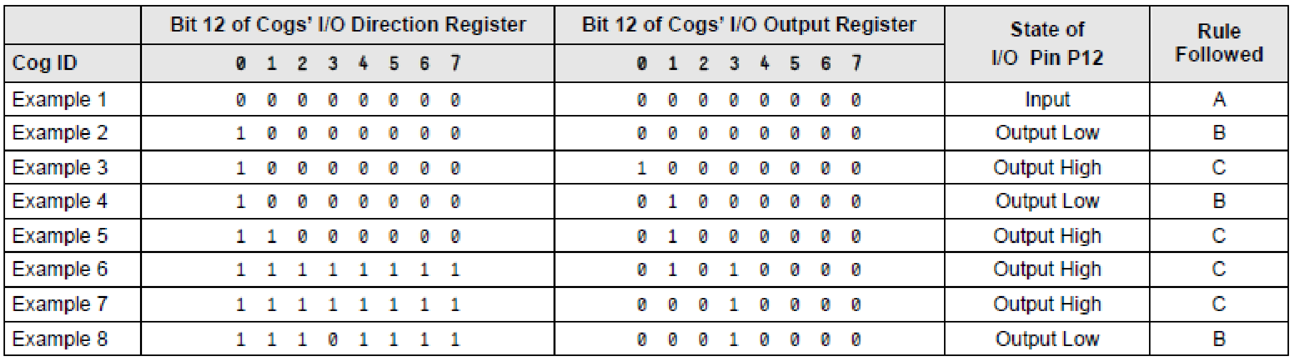
3.1.2 Clock and Execution Rate

The Propeller can be clocked using an internal oscillator or an external crystal or resonator. External clock speed can be set with flexible clock modes that are switchable at run-time. Using the internal 1x-16x PPL multiplier, the 5 MHz crystal included on the Demo Board has the effective range of 32 kHz to 80 MHz, with 80 MHz set as default. By decreasing the clock speed before a long "no operation" wait, and then increasing it again afterwards, the processor uses less power. Each cog executes 20 Million Instructions per Second (MIPS), giving an overall total of 160 MIPS with all cogs running.

3.1.3 I/O Pins

Each cog's I/O Output and Direction Registers influences the states of the Propeller's corresponding pins and are communicated through the entire cog collective by OR'ing the bits of each cog's I/O Direction Register together to become "Pin Directions" and OR'ing each cog's I/O Output Register together to become "Pin Outputs". A cogs Output state consists of the bits of its I/O modules (the Counters, Video Generator, and I/O Output Register) OR'd together and AND'd with the bits of its Direction register. All cogs can still access and influence I/O pins simultaneously, without electrical contention, by following these rules:

A. A pin is an input only if no active cog sets it to an output.
B. A pin outputs low only if all active cogs that set it to output also set it to low.
C. A pin outputs high if any active cog sets it to an output and also sets it high.



	Bit 12 of Cogs' I/O Direction Register								Bit 12 of Cogs' I/O Output Register								State of I/O Pin P12	Rule Followed
Cog ID	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7		
Example 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Input	A
Example 2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Output Low	B
Example 3	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	Output High	C
Example 4	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	Output Low	B
Example 5	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	Output High	C
Example 6	1	1	1	1	1	1	1	1	0	1	0	1	0	0	0	0	Output High	C
Example 7	1	1	1	1	1	1	1	1	0	0	0	1	0	0	0	0	Output High	C
Example 8	1	1	1	0	1	1	1	1	0	0	0	1	0	0	0	0	Output Low	B

Figure 8 – I/O Sharing Examples [XIX] **(Parallax Inc., 2011)**

3.1.4 Cog Memory Map

Each cog has 2KB of RAM, which are mostly general-purpose registers, with the last 16 registers reserved as special purpose registers. Cog RAM holds a copy of the Spin Interpreter when executing Spin code, or holds data, variables and program code when executing Propeller Assembler.

The special purpose registers are used to interface to the I/O pins, System Counter and local peripherals. When a cog is booted, locations 0 to 495 are loaded sequentially from Hub RAM / ROM and the special purpose registers are cleared to zero. Once loaded the cog begins executing instructions from location 0 of cog RAM and will continue until stopped or reset by either itself or another cog, or when a reset occurs.

Each Special Purpose Register may be accessed via:

1) its physical register address,
2) its predefined name, or
3) a register array variable with an index of 0 to 15.

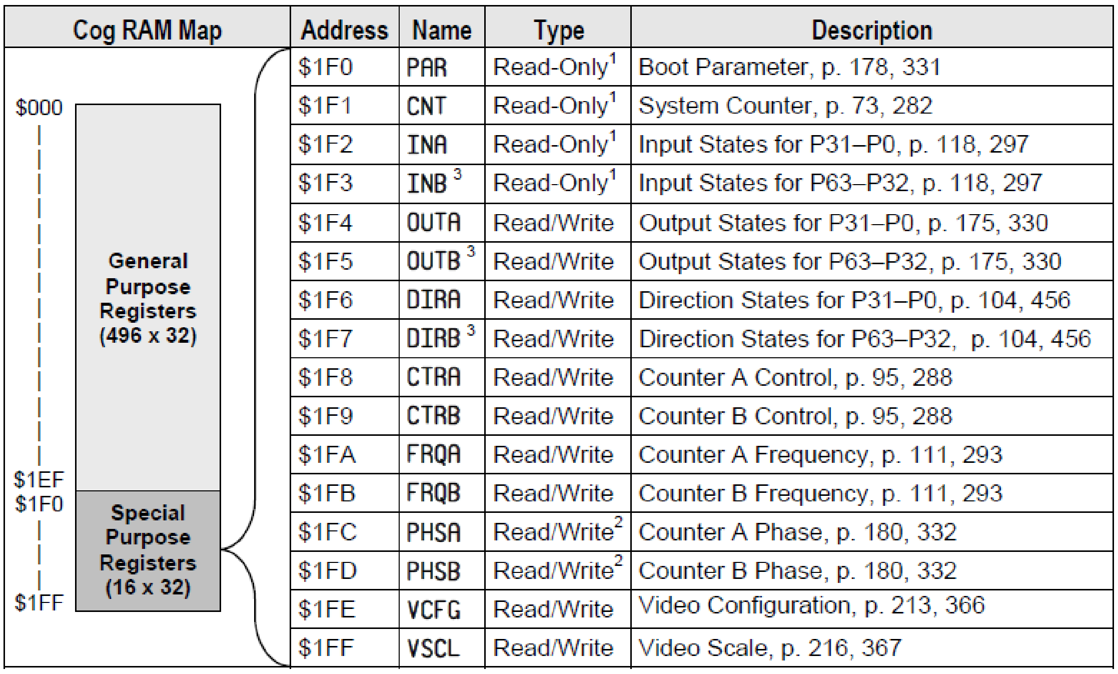
The following are examples in Propeller Assembly:

```
MOV  $1F4, #$FF                    'Set OUTA 7:0 high
MOV  OUTA, #$FF                    'Same as above
```

The following are examples in Spin:

```
SPR[$4] := $FF                     'Set OUTA 7:0 high
OUTA := $FF                        'Same as above
```

[XIV] **(Parallax Inc., 2006-2011)**


Cog RAM Map
$000
$1EF
$1F0
$1FF
General Purpose Registers (496 x 32)
Special Purpose Registers (16 x 32)

Address	Name	Type	Description
$1F0	PAR	Read-Only[1]	Boot Parameter, p. 178, 331
$1F1	CNT	Read-Only[1]	System Counter, p. 73, 282
$1F2	INA	Read-Only[1]	Input States for P31–P0, p. 118, 297
$1F3	INB [3]	Read-Only[1]	Input States for P63–P32, p. 118, 297
$1F4	OUTA	Read/Write	Output States for P31–P0, p. 175, 330
$1F5	OUTB [3]	Read/Write	Output States for P63–P32, p. 175, 330
$1F6	DIRA	Read/Write	Direction States for P31–P0, p. 104, 456
$1F7	DIRB [3]	Read/Write	Direction States for P63–P32, p. 104, 456
$1F8	CTRA	Read/Write	Counter A Control, p. 95, 288
$1F9	CTRB	Read/Write	Counter B Control, p. 95, 288
$1FA	FRQA	Read/Write	Counter A Frequency, p. 111, 293
$1FB	FRQB	Read/Write	Counter B Frequency, p. 111, 293
$1FC	PHSA	Read/Write[2]	Counter A Phase, p. 180, 332
$1FD	PHSB	Read/Write[2]	Counter B Phase, p. 180, 332
$1FE	VCFG	Read/Write	Video Configuration, p. 213, 366
$1FF	VSCL	Read/Write	Video Scale, p. 216, 367

Note 1: For Propeller Assembly, only accessible as a source register (i.e., `mov` *dest*, *source*). See the Assembly language sections for `PAR`, page 331; `CNT`, page 282, and `INA`, `INB`, page 297.

Note 2: For Propeller Assembly, only readable as a source register (i.e., `mov` *dest*, *source*); read modify-write not possible as a destination register. See the Assembly language section for `PHSA`, `PHSB` on page 332.

Note 3: Reserved for future use.

Figure 9 – Cog Memory Map [XIV] **(Parallax Inc., 2006-2011)**

3.2 Programming Languages

The Propeller natively runs two programming languages: Propeller Assembler & Spin. Chip Gracey designed both of these languages from scratch, combining well-known and brand new concepts.

To fully understand and utilise the tools and languages it is important to approach them with an open mind and not to let legacy-programming concepts and methods prevent the developer from exploiting the advantages of the Propeller. Parallax believes some concepts do not belong in true real-time processing environments because they tend to bring ‘turmoil’.

Propeller code is organised into objects. Objects are just programs written in a way that create self-contained entities to perform specific tasks and may be reused by other applications. Objects can be written entirely in Spin or can use various combinations of Spin and Propeller Assembler. It is possible to write objects almost entirely in PASM, but at least two lines of Spin are required to launch the object in a cog.

An object may consist of one or more other objects to build more sophisticated and complex applications. The top most object is referred to as the “Top Object File” and is the starting point for compiling Propeller Applications. The first public function in the Top Object File is what will be launched on cog 0 after boot.

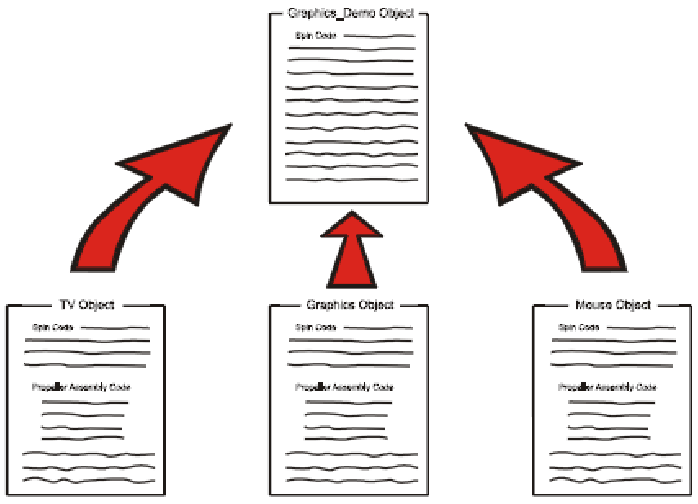


Figure 10 – Hierarchy of Top Object File and included objects

[XIV] **(Parallax Inc., 2006-2011)**

There are third party C compilers for the Propeller. ICCV7 by ImageCraft, which is proprietary and has been marked End of Life, a free ANSI C compiler called Catalina that is based on LCC, and also a port of GCC for the Propeller, which is in alpha and has known issues. One feature uses the pthread API to utilise multiple cogs.

There is also effort being made to run a Java Virtual Machine on the Propeller. Other languages supported are FORTH, BASIC and LISP. [X] **(Humanoido, 2009)**

Additional syntax definitions are in the Appendix.

3.2.1 Spin

Spin is a high level, object orientated, interpreted language and is the foundation for every Propeller Application. Each Propeller Object is a .spin file that has a hierarchical structure. Like Python, Spin uses indentation and whitespace to delimit blocks of code, instead of braces or keywords.

Spin code consists of six special purpose blocks. The list of block designators below are in the order they usually appear within Spin files.

- CON – Global Program Constants.
- VAR – Global Variables.
- OBJ – List of Objects to be included.
- PUB – Code for Public Spin subroutine.
- PRI – Code for Private Spin subroutine.
- DAT – Predefined data tables, memory reservations and Assembler code.

There can be multiple occurrences of each block type, arranged in any order, but there must be at least one PUB block declared per object. Typically, there is only one occurrence of CON, VAR, OBJ and DAT blocks, and multiple occurrences of PUB and PRI blocks.

The Propeller Compiler converts Spin code into byte code tokens that are interpreted at run-time by the chips built in Spin Interpreter. After boot, the primary cog is loaded with the Spin interpreter and begins reading byte code from Hub RAM. More than one interpreter can loaded into other cogs, so several Spin routines can be run concurrently. Because Spin is an interpreted language it runs slower than PASM, but can be more space efficient. PASM opcodes are 32 bits long while Spin directives are 8 bits long, and may be followed by a number of single bytes that specifies the directive’s behaviour.

Many Propeller Assembler commands have direct equivalents in Spin. This makes learning the two languages and using the chip relatively easy.

3.2.1.1 Memory Control

`LONG` is one of three multi-purpose declarations (`BYTE`, `WORD`, and `LONG`) that declares or operates on memory. `LONG` can be used to:

1) declare a long-sized (32-bit) symbol or a multi-long symbolic array in a `VAR` block, or
2) declare and initialize long-aligned, and/or long-sized, data in a `DAT` block, or
3) read or write a long of main memory at a base address with an optional offset.

Each example using `LONG` has `BYTE` and `WORD` equivalents

In `PUB` and `PRI` blocks, syntax 3 of `LONG` is used to read or write long-sized values of main memory. This is done by writing expressions that refer to main memory using the form: `long[`*BaseAddress*`][`*Offset*`]`. Here's an example.

```
PUB MemTest | Temp
       Temp := LONG[@MyData][1]                'Read long value
       long[@MyList][0] := Temp + $01234567    'Write long value

DAT
       MyData long 640_000, $BB50              'Long-sized/aligned data
       MyList byte long $FF995544, long 1_000  'Byte-sized/aligned
                                               'long data
```

`LONGMOVE` is used t copy longs from one region to another in main memory.

`LONGFILL` is used to fill longs of main memory with a value.

`LOOKUP` Get value at index (1..N) from a list.

`LOOKUPZ` Get value at zero-based index (0..N−1) from a list.

`LOOKDOWN` Get index (1..N) of a matching value from a list.

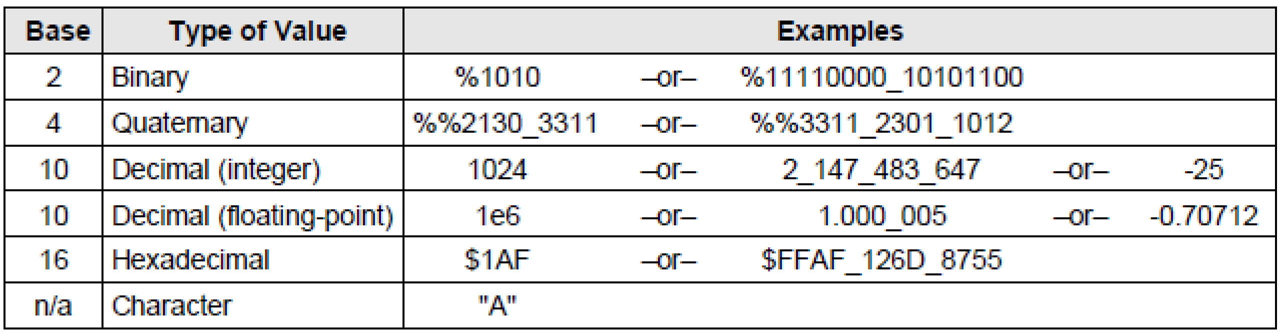
`LOOKDOWNZ` Get zero-based index (0..N−1) of a matching value from a list.

`STRSIZE` Get size of string in bytes.

`STRCOMP` Compare a string of bytes against another string of bytes.

3.2.1.2 Value Representation

Parallax decided to implement their own convention for representing binary, quaternary, decimal, hexadecimal values and character formats. Numerical values can use underscores as group separators for clarity. Separators can be used instead of commas (in decimal values) or to form logical groups such as nibbles, bytes and words, etc.



Base	Type of Value	Examples				
2	Binary	%1010	–or–	%11110000_10101100		
4	Quaternary	%%2130_3311	–or–	%%3311_2301_1012		
10	Decimal (integer)	1024	–or–	2_147_483_647	–or–	-25
10	Decimal (floating-point)	1e6	–or–	1.000_005	–or–	-0.70712
16	Hexadecimal	$1AF	–or–	$FFAF_126D_8755		
n/a	Character	"A"				

Figure 11 – Digital Value Representation [XIV] **(Parallax Inc., 2006-2011)**

3.2.1.3 Comments

The Propeller syntax supports four different types of comments.

`'...`	– Single-line code comment (apostrophe).
`''...`	– Single-line document comment (two apostrophes, NOT a quotation mark).
`{...}`	– Multi-line code comment (curly braces).
`{{...}}`	– Multi-line document comment (two curly braces).

An example from the Programming Tutorial is below:

```
{{Output.spin
Toggles Pin with Delay clock cycles of high/low time.}}
CON
  Pin = 16                    { I/O pin to toggle on/off }
  Delay = 3_000_000           { On/Off Delay, in clock cycles}

PUB Toggle
''Toggle Pin forever
{Toggles I/O pin given by Pin and waits Delay system clock cycles
in between each toggle.}
  dira[Pin]~~                 'Set I/O pin to output direction
  repeat                      'Repeat following endlessly
    !outa[Pin]                ' Toggle I/O Pin
    waitcnt(Delay + cnt)      ' Wait for Delay cycles
```

[XI] **(Parallax Inc., 2006)**

If a comment begins with one apostrophe or one curly brace then it is a comment meant to be read by a developer reviewing the source code.

If a comment begins with two apostrophes or two curly braces then it is a special type of comment that can be extracted by the Propeller Tool into a document format containing no source code for easier reading.

[XIV] **(Parallax Inc., 2006-2011)**

3.2.2 Propeller Assembler

Propeller Assembler is a low level, highly optimized language that is executed in its pure form at run-time.

When a Propeller Application initially boots up, only Spin code is executed. At any time, however, that Spin code can choose to launch assembly code into a cog of its own. The `COGNEW` and `COGINIT` commands are used for this purpose. The following Spin code example launches the `Toggle` assembly code shown below.

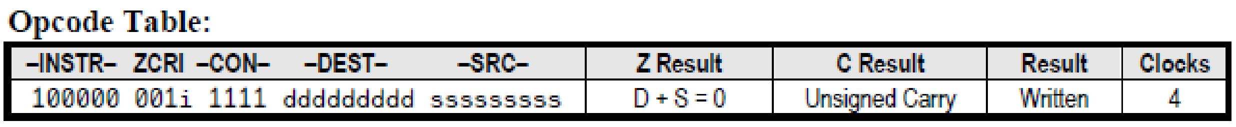
```
PUB Main
    cognew(@Toggle, 0)                    'Launch Toggle code
```

A Data Block is a section of source code, beginning with the keyword `DAT`, which contains Propeller Assembly code, pre-defined data and memory reserved for run-time use. Assembly code and data can be intermixed, if necessary, so that data is loaded into a cog along with the assembly code.

3.2.2.1 Opcodes and Operands

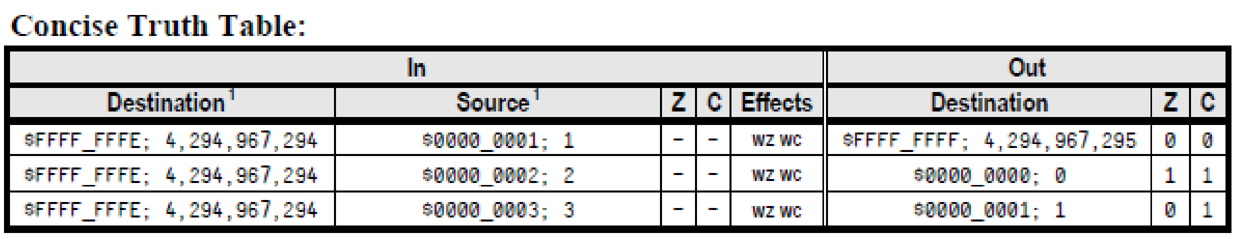
Most instructions have two data operands; a destination value and a source value. For example, the format for an `ADD` instruction is: `add` *destination*, <#> *source*

The *destination* operand is the 9-bit address of a register containing the desired value to operate on. The *source* operand is either a 9-bit literal value (constant) or a 9-bit address of a register containing the desired value. The meaning of the source operand depends on whether or not the literal indicator "#" was specified.



Opcode Table:

-INSTR- ZCRI -CON- -DEST- -SRC-	Z Result	C Result	Result	Clocks
100000 001i 1111 ddddddddd sssssssss	D + S = 0	Unsigned Carry	Written	4



Concise Truth Table:

In					Out		
Destination[1]	Source[1]	Z	C	Effects	Destination	Z	C
$FFFF_FFFE; 4,294,967,294	$0000_0001; 1	-	-	wz wc	$FFFF_FFFF; 4,294,967,295	0	0
$FFFF_FFFE; 4,294,967,294	$0000_0002; 2	-	-	wz wc	$0000_0000; 0	1	1
$FFFF_FFFE; 4,294,967,294	$0000_0003; 3	-	-	wz wc	$0000_0001; 1	0	1

Figure 12 – Opcode and Truth Table for `ADD` Instruction

The opcode table's first column contains the Propeller Assembly Instruction opcode, consisting of the following fields:

- **INSTR** (bits 31:26) - Indicates the instruction being executed.
- **ZCRI** (bits 25:22) - Indicates instruction's effect status and **SRC** field meaning.
- **CON** (bits 21:18) - Indicates the condition in which to execute the instruction.
- **DEST** (bits 17:9) - Contains the destination register address.
- **SRC** (bits 8:0) - Contains the source register address or 9-bit literal value.

The bits of the **ZCRI** field each contain a 1 or 0 to indicate whether or not the 'Z' flag, 'C' flag, and 'R'esult should be written, and whether or not the **SRC** field contains an 'I'mmediate value (rather than a register address). The Z and C bits of the **ZCRI** field are clear (0) by default and are set (1) if the instruction was specified with a WZ and/or WC effect The R bit's default state depends on the type of instruction, but is also affected if the instruction was specified with the WR or NR effect. The I field's default state depends on the type of instruction and is affected by the inclusion, or lack of, the literal indicator (#) in the instruction's source field.

The bits of the **CON** field usually default to all ones (1111) but are affected if the instruction was specified with a condition.

3.2.2.2 IF_x (Conditions)

Every Propeller Assembly instruction has an optional "condition" field that is used to dynamically determine whether or not it executes when it is reached at run time.

This feature simplifies critical timing routines; enabling multi-decision code to execute along the same path, taking the same amount of time, regardless of the decision. This makes applications less prone to jitter during conditional execution since less jumping is involved. It also helps to keep code easily readable and understood.

This example is from the microphone to VGA demo code:

```
              cmp       mode,#0                wz       'wait for negative trigger threshold
if_z          cmp       asm_sample,trig_min    wc
if_z_and_c    mov       mode,#1
if_z          jmp       #:loop

              cmp       mode,#1                wz       'wait for positive trigger threshold
if_z          cmp       asm_sample,trig_max    wc
if_z_and_nc   mov       mode,#2
if_z          jmp       #:loop
```

Instructions that modify a value and possibly jump, based on the result, require a different amount of clock cycles depending on whether or not a jump is required. These instructions take 4 clock cycles if a jump is required and 8 clock cycles if no jump is required. Since loops utilizing these instructions typically need to be fast, they are optimized in this way for speed.

3.2.2.3 REV Instruction

REV Value, <#> Bits

REV (Reverse) reverses the lower (32 - Bits) of Value's LSB and clears the upper Bits of Value's MSBs. This instruction is particularly useful during the initial stages of the FFT algorithm. The index for the input array is rearranged to implement the butterfly of multiples and additions.

[XIV] **(Parallax Inc., 2006-2011)**

3.2.2.4 Code Syntax Definition

DAT

<***Symbol***> <***Condition***> ***Instruction*** ***<Effect(s)>***

- ***Symbol*** is an optional name for the data, reserved space, or instructions that follow.
- ***Condition*** is an assembly language condition, IF_C, IF_NC, IF_Z, etc.
- ***Instruction*** is an assembly language instruction, ADD, SUB, MOV, etc., and all its operands.
- ***Effect(s)*** is/are one, two or three assembly language effects that cause the result of the instruction to be written or not, NR, WR, WC, or WZ.

The following example toggles pin 0 every ¼ second.

```
DAT
            org       0                 'Reset assembly pointer
Toggle      rdlong    Delay, #0         'Get clock frequency
            shr       Delay, #2         'Divide by 4
            mov       Time, cnt         'Get current time
            add       Time, Delay       'Adjust by 1/4 second
            mov       dira, #1          'set pin 0 to output
Loop        waitcnt   Time, Delay       'Wait for 1/4 second
            xor       outa, #1          'toggle pin
            jmp       #Loop             'loop back

Delay res 1
Time res 1
```

A note on variables in Propeller Assembly:

The assembly example above includes two symbols used as variables, Delay and Time. Since these are defined as reserved memory they are not initialized; upon launch of the assembly code, they will contain whatever value happened to be in the register they occupy in Cog RAM. Assembly variables are defined this way when it is the intention of the code to initialize them during run time.

To create a compile-time, initialized assembly variable instead, use the DAT block's LONG data declaration. For example:

```
Delay long 1
```

It can be used in place of RES, and upon launch of the assembly code, will load the value following it into the register that the variable occupies in Cog RAM.

It is important to use RES only after the final instructions and data in a logical assembly program. Placing RES in a prior location could have unintended results.

Assembly instructions and data are placed in the application memory image in the exact order entered in source, independent of the assembly pointer. This is because launched assembly code must be loaded in order, starting with the designated routine label. However, since RES does not generate any data (or code), it has absolutely no effect on the memory image of the application; RES only adjusts the value of the assembly pointer.

By nature of the RES directive, any data or code appearing after a RES statement will be placed immediately after the last non-RES entity, in the same logical space as the RES entities themselves.

If using a mix of LONG-based and RES-based symbols, make sure to define all RES-based symbols last!

3.2.2.5 Data Declaration Definition

DAT

<Symbol> Alignment <Size> <Data> <[Count]> <, <Size> Data <[Count]>>

- ***Symbol*** is an optional name for the data, reserved space, or instruction that follows.
- Alignment is the desired alignment and default size (BYTE, WORD, or LONG) of the data elements that follow.
- ***Size*** is the desired size (BYTE, WORD, or LONG) of the following data element immediately following it; alignment is unchanged.
- ***Data*** is a constant expression or comma-separated list of constant expressions.
- Quoted strings of characters are also allowed; they are treated as a comma-separated list of characters.
- ***Count*** is an optional expression indicating the number of byte-, word-, or long-sized entries of *Data* to store in the data table.

You can also use the BYTE, WORD, and LONG declarations to read main memory locations. For example:

```
DAT
       MyData      byte $FF[10], 25, %1010

PUB GetData | Temp
       Temp := BYTE[@MyData][0] 'Get first byte of data table
```

[XIV] **(Parallax Inc., 2006-2011)**

3.2.3 The Hub

One of the biggest limitations of the Propeller is its relatively small amount of RAM. The Hub contains 32KB of globally accessible RAM, which contains program and variable data. Each cog has 2KB of RAM that is overwritten from Hub RAM on cog launch.

To maintain system timing each cog has access to the Hub, Round Robin fashion, for 2 system clock cycles. During that window, the cog can initiate a Hub Instruction.

Propeller Assembly hub instructions include `CLKSET`, `COGID`, `COGINIT`, `COGSTOP`, `HUBOP`, `LOCKCLR`, `LOCKNEW`, `LOCKRET`, `LOCKSET`, `RDBYTE`, `RDLONG`, `RDWORD`, `WRBYTE`, `WRLONG` and `WRWORD`.

Hub instructions take 8 cycles to complete. Most cog instructions take four clock cycles to complete, so efficient Hub Access timing can be achieved by interleaving single Hub instructions with two cog instructions Consult the Propeller Manual for information on specific instruction execution time.

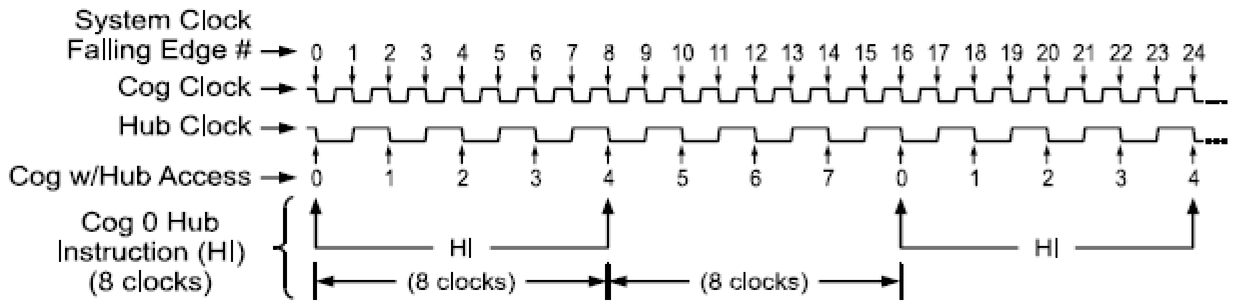


Figure 13 – Cog-Hub Interaction – Best Case Scenario

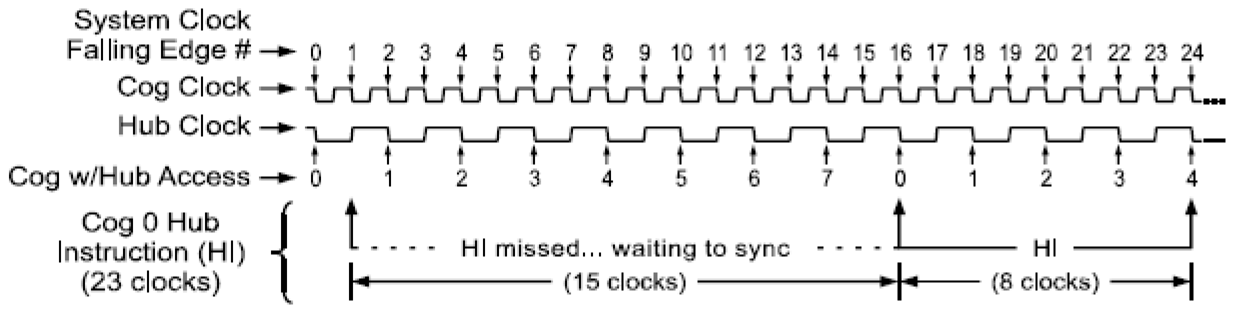


Figure 14 – Cog-Hub Interaction – Worst Case Scenario

[XIX] **(Parallax Inc., 2011)**

3.2.4 Floating Point

The Library included with the Propeller Tool provides a complete Floating Point implementation. Since the Propeller does not include a Floating Point Unit, Floating Point functions are effectively implemented by distributing the work amongst other cogs. The list below is from the comment heading of Float32Full.spin

	FloatMath	Float32	Float32Full
Cogs required:	0	1	2
Execution Speed: e.g. FADD (usec)	Slow 371	Fast 39	Fast 39
Methods: FAdd, FSub, FMul, FDiv FFloat, FTrunc, FRound FSqr, FNeg, FAbs	 • • •	 • • •	 • • •
Sin, Cos, Tan Radians, Degrees Log, Log10, Exp, Exp10 Pow, Frac FMod Fmin, Fmax		• • • • • •	• • • • • •
FMod ASin, ACos, ATan ATan2 Floor, Ceil FFunc			• • • • •

The Parallax store offers a 32-bit Floating Point Coprocessor for a reasonable $20. The uM-FPU V3.1 provides support for 32-bit IEEE 754 floating point operations and 32-bit long integer operations, with both SPI and I^{2}C support. The uM-FPU feature set boasts FFT operations, which would be ideal for this project.

3.3 Tool chain

There are varieties of ways to program the Propeller. Two of the most popular Integrated Development Environments (IDEs) are The Propeller Tool and Brad’s Spin Tool. Both of these tools are intelligent editors that perform the same basic tasks but with minor differences.

The Propeller Tool is the IDE designed for Windows XP provided by Parallax for developing Propeller applications. Brad’s Spin Tool is a cross-platform tool suite with binaries provided for Windows 95 - XP, Linux 2.4 up and Mac OSX.

Both of the tools automatically colour the background of different block types, and even consecutive occurrences of the same type, to make it easy to identify the beginning and end of blocks. This functionality is intended to solve a common problem; as projects get larger it becomes harder to find a particular method or function quickly. Colour coding the background eliminates the need for users to waste time manually inserting text separators.

The Propeller language and IDE integration offers a feature rich, interactive and easy to use interface for rapid application development.

Development for Eclipse integration is in progress and the upcoming Propeller 2 is rumoured to be released with an Eclipse plug-in instead of another complete custom IDE.

The Propeller Tool and bst are custom made for a single platform. Compared with commercial IDEs such as Visual Studio or Eclipse, many of the expected features are lacking, such as direct debug support and integrated console output. However, they do offer all other required functionality with a simple and compact feel.

This section shall explore some of the details and difference between the Propeller Tool and Brad’s Spin Tool.

3.3.1 The Propeller Tool

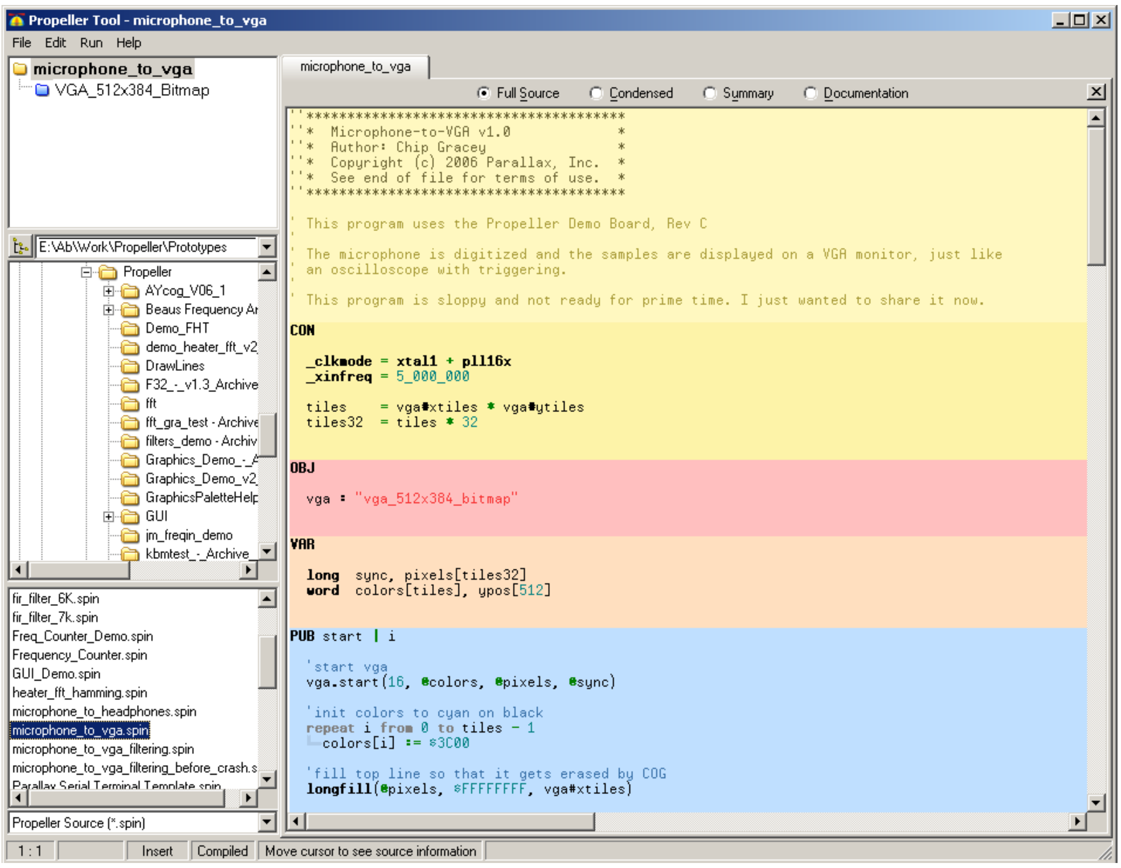
The Propeller Tool was designed by Parallax to run on Microsoft Windows 2000, XP, Vista and Windows 7. It is available from the Parallax Website, and is included on the CD with the Demo Board. The Propeller Tool includes the Parallax font, Parallax Serial Terminal, Propeller Library and a USB Future Technology Devices International (FTDI) driver for Windows to recognise the Propeller USB Serial Converter. When starting the Propeller Tool for the first time, the user is asked to associate .spin and .eeprom files with the tool.

3.3.1.1 View Modes

The Propeller Tool has a number of view modes to easily navigate source files. The IDE is split into four sections called panes. Each has a specific function. The top left pane is the Object View Pane that displays the object hierarchy of the most recently successfully compiled project.

The central left pane is the Recent Folders field and Folders List Pane. The Folder List displays the hierarchical view of the folders within each drive. The Recent Folders field provides a drop-down list of special folders (Propeller Library and Propeller Demos) and the folders containing recently opened files. The button to the left of the drop-down menu toggles between displaying the folders on the entire drive and just the drives and folders recently used.

The bottom left pane is the File List and Filter field Pane. The File List displays all the files in the selected folder that match the criteria of the Filter field, which is a list of file extensions.



Full Source [✓] Condensed Summary Documentation

```
''***************************************
''*  Microphone-to-VGA v1.0             *
''*  Author: Chip Gracey                *
''*  Copyright (c) 2006 Parallax, Inc.  *
''*  See end of file for terms of use.  *
''***************************************

' This program uses the Propeller Demo Board, Rev C
'
' The microphone is digitized and the samples are displayed on a VGA monitor, just like
' an oscilloscope with triggering.
'
' This program is sloppy and not ready for prime time. I just wanted to share it now.

CON

  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

  tiles    = vga#xtiles * vga#ytiles
  tiles32  = tiles * 32

OBJ

  vga : "vga_512x384_bitmap"

VAR

  long  sync, pixels[tiles32]
  word  colors[tiles], ypos[512]

PUB start | i

  'start vga
  vga.start(16, @colors, @pixels, @sync)

  'init colors to cyan on black
  repeat i from 0 to tiles - 1
    colors[i] := $3C00

  'fill top line so that it gets erased by COG
  longfill(@pixels, $FFFFFFFF, vga#xtiles)
```

Figure 15 – The Propeller Tool

The right pane is the Editor Pane. This displays the source code files selected from the File List pane.

Each file opened is organised within the Editor Pane in a new tab. The currently active tab is highlighted differently from the rest. Tabs can be rearranged in the Editor Pane by dragging them; this can be used to split the active window into two or more sections. Tabs dragged outside the Editor Pane will open in their own window.

The Editor Pane has four different source code display views: Full Source, Condensed, Summary and Documentation. These modes can be changed by selecting the respective radio button at the top of the Pane. The Documentation view cannot be accessed if the object cannot be compiled.

Full Source view displays every line of source code in the file. Condensed view hides every line that contains only comments and white space, showing only capable lines. Summary view displays only block headings in a convenient way to see the structure of an object at a glance. When a block is clicked the view is expanded back to Full Source view. Documentation view displays the documentation generated by the compiler from the source code’s comments, along with some statistics about the compiled code, e.g. number of Longs taken up by Program and Variable data.

Line numbers can optionally be displayed. It is also possible to place bookmarks on various line numbers to jump quickly to a desired location.

3.3.2 Brad’s Spin Tool (bst)

Created by Brad Campbell / Viridian Consulting, bst (never capitalised) is a cross-platform tool suite that runs on i386 Windows 95-XP, i386 Linux and Mac OSX 10.4 - 10.6.

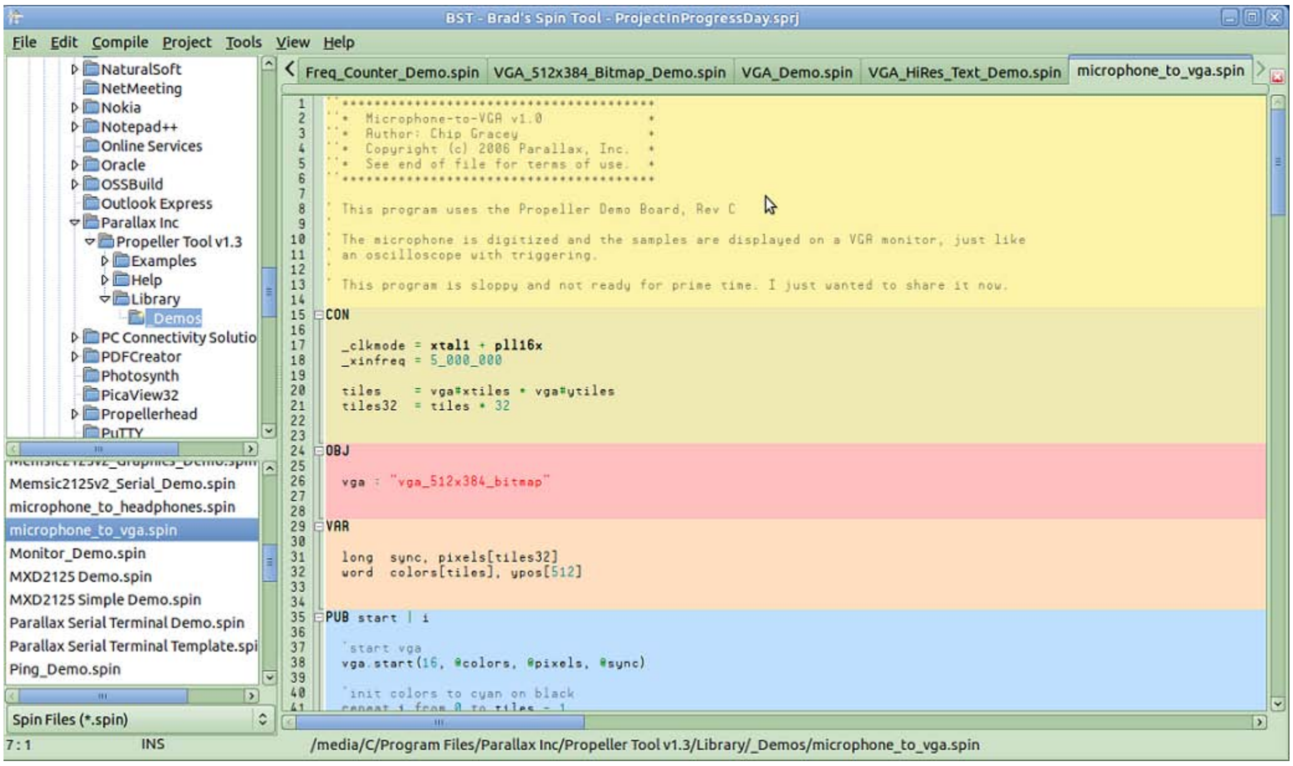
The suite consists of:

bstl: The Propeller Loader: A command line application that allows the user to load pre-compiled .binary or .eeprom images into the Propeller.

bstc: The Spin Compiler: A completely Parallax compliant SPIN and PASM compiler and linker, which also includes bstl.
Extra features include being able to emit a list file, some basic optimisation, zip file generation and extensions to the Official Spin Compiler, such as C like condition compilation (#define).

bst: The IDE: A complete windows IDE that aims to be work-alike and comparable to the Propeller Tool. It has a basic serial terminal built in that allows copy and pasting, as well as saving to a log file.

A prerequisite of bst is the installation of a suitable font. A modified version of the Parallax Propeller Font is provided online.



```
''*************************************
''*  Microphone-to-VGA v1.0           *
''*  Author: Chip Gracey              *
''*  Copyright (c) 2006 Parallax, Inc.  *
''*  See end of file for terms of use.  *
''*************************************

' This program uses the Propeller Demo Board, Rev C
'
' The microphone is digitized and the samples are displayed on a VGA monitor, just like
' an oscilloscope with triggering.
'
' This program is sloppy and not ready for prime time. I just wanted to share it now.

CON

  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

  tiles    = vga#xtiles * vga#ytiles
  tiles32  = tiles * 32


OBJ

  vga : "vga_512x384_bitmap"


VAR

  long  sync, pixels[tiles32]
  word  colors[tiles], ypos[512]


PUB start | i

  'start vga
  vga.start(16, @colors, @pixels, @sync)

  'init colors to cyan on black
  repeat i from 0 to tiles - 1
```

Figure 16 – bst Linux IDE

3.3.3 Comparison of bst and Propeller Tool

Some of the features provided by bst that are not available on the Propeller tool are:

<u>The list file</u>: bst produces a comprehensive list of all generated SPIN code and outputs a human readable look into the byte code and structures generated by the code.

<u>The serial terminal behaviour</u>: The Propeller Tool’s included serial terminal is required to be disconnected while attempted to program the Propeller. After compilation and download is complete, the user must manually switch to the terminal window and press connect again before output is received. This can be tricky when the serial message is sent soon after boot and the user is racing to connect in time to see it. The connection state of the bst serial terminal is automatically managed so that the user does not need to disconnect/reconnect during code download. It is always ready to receive.

There are some control characters used by the Propeller Tool’s serial terminal (e.g. clear screen and return home) that bst’s serial terminal doesn’t handle, displaying question marks instead of the expected behaviour.

<u>Compiler optimization options</u>: The bstc has a number of optimizations built in to enhance SPIN interpretation speed (where possible), make code smaller and intelligently remove unused portions of code. One of the advantages of the Propeller is the “off the shelf” objects that are available to be plugged into a new project. One of the down sides is that the object may contain unused code. The compiler looks at the code paths and leaves out methods that are never called, resulting in significant size saving.

Complete Project Management: The bst IDE allows the user to save all open tabs, their line positions and the file and directory listing windows, into a project file that remembers all of this information. This is a very useful feature, since the Propeller Tool does not remember open tabs between sessions.

Comprehensive error reporting: The bstc produces a list of warnings, errors and information on unused variables and redundant objects. It also lists user defined reports from the code (#error/#warn/#info). Clicking on the error in the display box will take the Editor to the offending line.

Management of Multiple Propellers: bst allows multiple Propellers to be connected at the same time, with management of which Top Object File belongs to which Propeller. This information is saved in the project file and allows easy and seamless management of multiple Propeller projects. [I] **(Campbell, 2009)**

Some features of the Propeller Tool that are not available on bst are:

Bookmarks: The Propeller Tool allows the developer to bookmark particular lines and code blocks to jump quickly to a location.

Block Group Indicators: The Propeller Tool provides visual aids for determining how code is influenced by whitespace indentation. For example, the instructions following an if statement only belong to the statement when they are indented.

[XX] **(Parallax Inc., 2011)**

3.4 Support for the Propeller – Parallax Community

The Parallax online community is very active and collaborative. There are multiple sections of the website dedicated to various aspects of user interaction.

The Object Exchange (ObEx) is where users can submit objects to Parallax for approval and are subsequently available for download. There are many thousands of objects available here for free download and use.

The forums are available for help and discussion on any aspects of the hardware and software Parallax provides and supports. There is also a Wiki.

The main website lists everything Parallax provides, including lists of local distributors of their hardware. The nearest location to Bristol is Active Robots, based in Somerset. An online shop for Propeller kits and equipment is www.spinvent.co.uk based in Norwich.

3.5 Propeller Demo Board

The Propeller Demo Board demonstrates the Propeller's varied capabilities. This will be used as the target platform for the project.

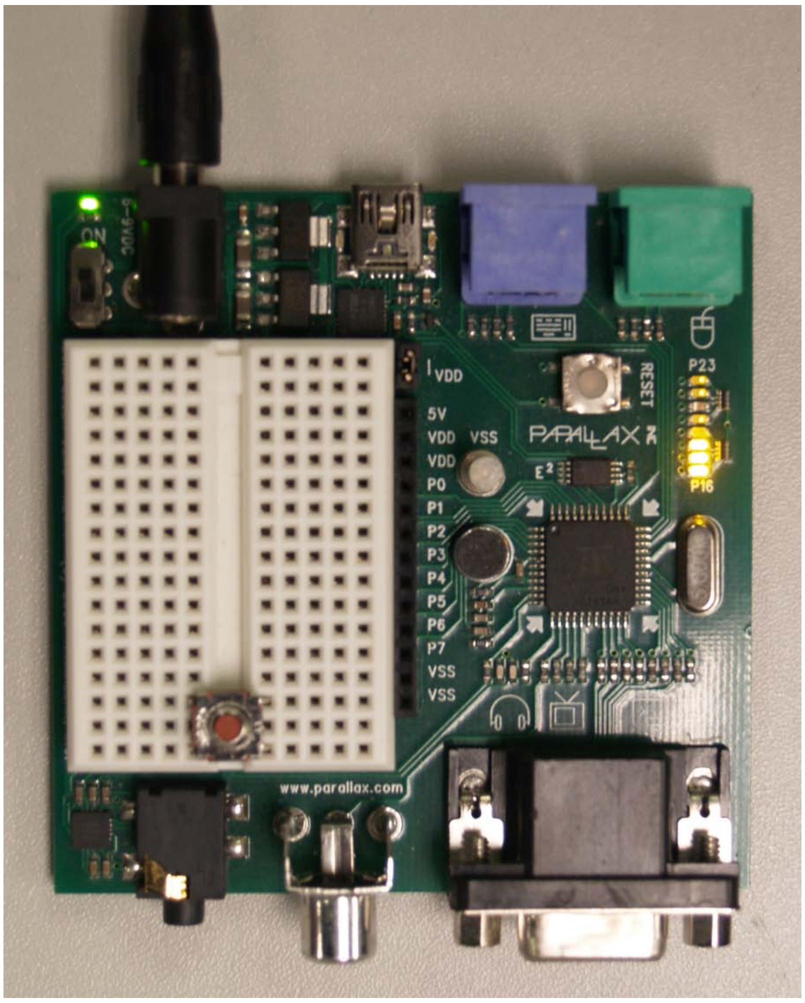


Figure 17 – Propeller Demo Board – Top View

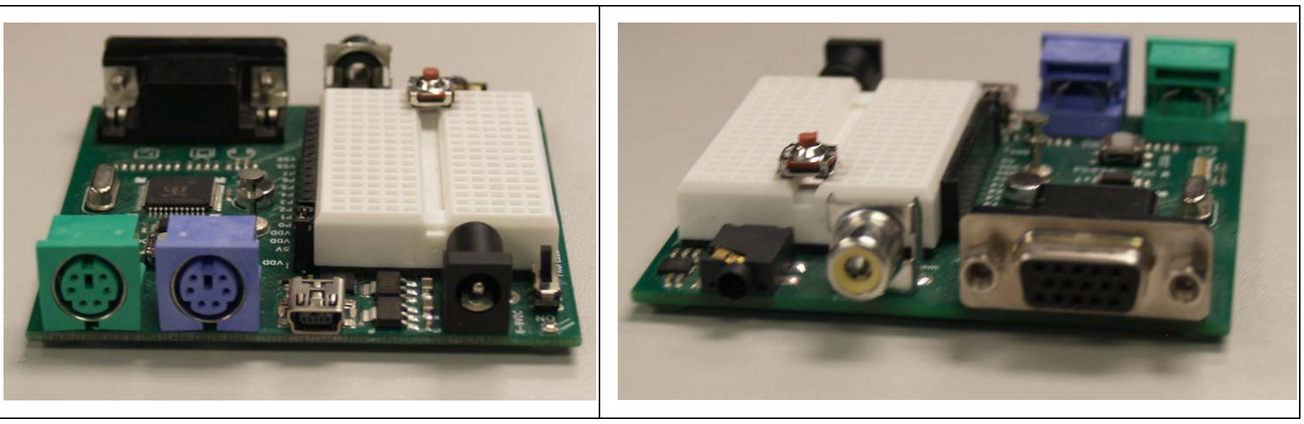


Figure 18 – Propeller Demo Board – Side Views

The Propeller Demo Board features:

- P8X32A-Q44 Propeller Chip
- 24LC256-I/ST EEPROM for program storage
- Replaceable 5.000MHz crystal
- 3.3V and 5V regulators with on/off switch
- USB-to-serial interface for loading and communication
- VGA output
- TV output
- Stereo output with 16-ohm headphone amplifier
- Electret microphone input
- Two PS/2 connectors for mouse and keyboard I/O
- 8 LEDs (share VGA pins)
- Pushbutton for reset
- Big ground post for scope hookup
- I/O pins P0-P7 are free and brought out to header
- Breadboard for custom circuits

[XV] **(Parallax Inc., 2007)**

The variety of hardware available on the Demo Board makes it ideal for an educational environment. Low cost, low part count components that interface easily with existing standards highlights the Propeller's versatility.

There are thousands of code examples and demos available for the demo board from the Parallax Object Exchange and the forums.

3.5.1 Debugging

There is an official debugger available from Parallax that they offer on their website for a free 30 day trial.

Another debugger tool has been written by Andy Schenk, from Insonix, called Propeller Assembler Source-code Debugger (PASD). The debugger runs on a separate cog and communicates with the host via its own software.

A special section of hex encoded instructions is placed at the beginning of the Propeller Assembler code that the user wishes to debug. This enables the debugger to take control of the running cog to enable setting break points.

The Propeller does not contain hardware that supports breakpoints. Instead, it is necessary to simulate breakpoints by temporarily modifying target instructions.

My experience with the debugger is that it does stop at break points as expected, and allows the user to single step through assembler instructions; however, it does not allow the user to enter a break point again after execution has been resumed. Execution will only step the first time a breakpoint is reached. The user can step through and watch values change, but the cog will not resume normal execution when running from the debugger. This is extremely tedious when trying to debug routines with large loops.

Using the Parallax Serial Terminal (pst) object, debugging information can be sent to the host via the serial terminal.

3.5.2 Propeller Demo Library

Included with the Propeller Tool is a library of examples and demos for the Demo Board, mainly written by the Propeller’s designer, Chip Gracey. The demo that sparked the inspiration for this project is called ‘microphone_to_vga.spin’. It runs with two cogs. A microphone sampler object plots to a pixel array and a VGA output object displays the array. The display shows the sampled waveform, similar to an oscilloscope screen. This demo was modified to build a sampler object that writes sample blocks to Hub Memory.

There are a few different styles of graphical display objects, depending on the required level of functionality. The standard theme for VGA objects is an array of Long aligned tiles. Each tile is a one bit per pixel bitmap with colour assigned from a palette that lists the two colours to be represented by 0 or 1 for each pixel. Tiles are defined as certain heights and widths appropriately.

Some VGA demonstrations utilise the Parallax Font stored in Hub ROM to display text. Using a single cog for generating graphics means the screen resolution is quite low. One cunning demo shows the power of distributing each quarter of the display to separate cogs and allowing reasonably high resolution for a terminal demo, with keyboard and mouse input.

Another set of graphic demos uses the Propeller’s Composite TV hardware as output and generates fairly complex and visually active displays.

4 Research

This chapter discusses the topics discovered during system development.

4.1 Audio Analysis

4.1.1 Frequency Counter

One of the most common implementations for determining the frequency of a signal is to count the number of times the amplitude of a wave crosses zero of the Y axis in a given time. This method is relatively simple to implement using the existing code library for the Propeller.

This method is very accurate for waveforms that have continuous wavelength and amplitude. One of the key characteristics of musical instruments is the fluctuations of their waveforms. This gives rise to different instruments' timbre, for example a trumpet and guitar playing the same note at the same volume will be noticeably different from each other to the human ear. This is due to the particular variations of the waveform produced by the medium used to create it, in this case brass and string. These can be reliant on a number of factors, such as the resonance of the instrument due to its shape.

Notes are usually louder as they are produced and then decay over time. The initial hit of a note is known as the attack. When measuring the frequency of a musical note, the attack may come in the middle of a given sample window.

The Microphone to VGA demo uses a 512 by 384 pixel array. The VGA object uses a two colour palette and each pixel is represented by one bit. The display is updated from left to right by XORing a single pixel. A record of which pixel has been changed is stored in an array and the index is then shifted one pixel to the right. As microphone values are evaluated, every time a maximum or minimum threshold is reached and the wave begins to change direction, the previous pixel is removed and a new pixel is drawn to represent the amplitude reached by the captured value. Every 512 direction changes the screen is fully updated and the index starts back on the left side of the screen.

Since the threshold values that determine when a wave is changing direction are dynamically updated, it is possible for a wave to change direction before crossing zero of the Y axis, due to the irregularities of notes due to resonance of instruments, as mentioned earlier. For this reason it is not enough to rely on the number of direction changes to accurately count the frequency of the signal. An explicit check for the number of zero crossings is required.

If the number of zero crossings within the visible screen is counted, and the time taken is known, then the frequency of the signal within that window can be extrapolated with a reasonable degree of accuracy.

4.1.2 Note Detection

The Nyquist–Shannon sampling theorem states that highest frequency that can be detected when sampling is half that of the sample rate. A higher frequency appearing as lower frequencies is known as aliasing.

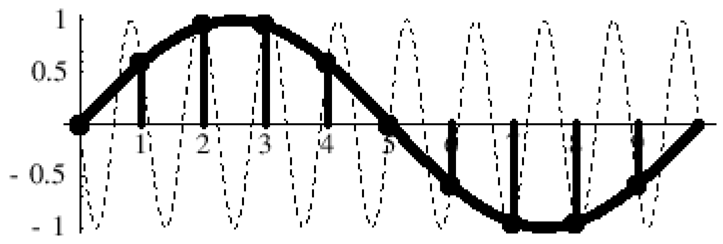


Figure 19 – Example of frequency-domain aliasing due to undersampling in the time domain.

[XXII] **(Smith, 2007)**

Accurate detection of the highest desired frequency implies the sampling rate should be more than double said frequency. For the purposes of this project the highest desired frequency is 1500 Hz. As such, 4 KHz shall be set as the minimum sampling rate.

The equation for calculating the conventional Western letter representation of music notes is:

$$f = 2^{k(n/12)} \times 440\ \text{Hz}$$

Where *f* is the frequency, *k* is the octave relative to A4 and *n* is the given offset of semitones from A4 in the following set:

A, A#, B, C, C#, D, D#, E, F, F#, G, G#

Using this equation is it possible to calculate the note, octave, and percentage (cents) offset from an arbitrary input frequency.

$$n = \log_2(\frac{f}{440}) * 12$$

The octave is the truncated value of *n* + 4, since A4 is the baseline note (440 Hz). This needs to be offset again since the convention for changing octave number occurs on the C note.

Cents are calculated by multiplying *n* by 100 and offsetting values fewer than 100 by 50.

The human perception of musical note progression is logarithmically spaced. The difference between a note in one octave the next octave is exactly double its own frequency.

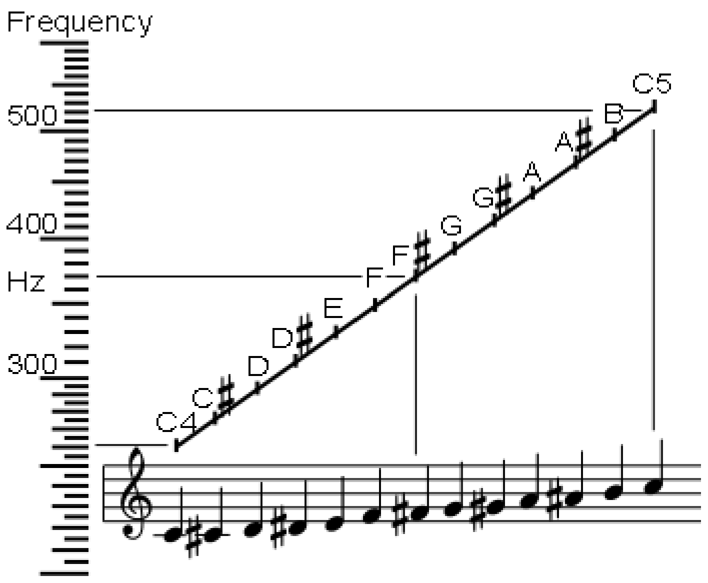


Figure 20 – Frequency vs Name **(Wikipedia, 2008)**

4.1.3 Low-Pass Filter

To prevent aliasing of higher frequencies than the desired range for analysis, a low-pass filter is required. This will eliminate any frequencies above a chosen cut off value. Traditionally, Finite Impulse Response (FIR) filters are provided by hardware using capacitors, however it is possible to simulate the same functionality in software.

The low-pass filter is one of a set of functions available to remove unwanted frequency ranges of a given signal. Others include band-pass filters, high-pass filters and band-stop filters.

4.2 The Fourier Transform

4.2.1 The Fourier series

Joseph Fourier was a French mathematician and physicist who, in 1822, published work describing conductive heat dissipation on a square metal plate. This work put forward the idea of modelling complicated heat sources as a superposition of simple sine and cosine waves. He is credited for beginning many different branches of mathematics, including the Fourier series, Fourier analysis, Fourier synthesis, Fourier's theorem and the Fourier transform.

The Fourier series is defined as:

> an infinite trigonometric series of the form $\frac{1}{2}a_0 + a_1\cos x + b_1\sin x + a_2\cos 2x + b_2\sin 2x + \ldots$, where $a_0, a_1, b_1, a_2, b_2 \ldots$ are the **Fourier coefficients**. It is used, esp in mathematics and physics, to represent or approximate any periodic function by assigning suitable values to the coefficients
>
> [XXIII] **(William Collins Sons & Co. Ltd., 2009)**

Alternatively:

> **Fourier series,** In mathematics, an infinite series used to solve special types of differential equations. It consists of an infinite sum of sines and cosines, and because it is periodic (i.e., its values repeat over fixed intervals), it is a useful tool in analyzing periodic functions. Though investigated by Leonhard Euler, among others, the idea was named for Joseph Fourier, who fully explored its consequences, including important applications in engineering, particularly in heat conduction.
>
> [V] **(Encyclopædia Britannica Online, 2012)**

Fourier’s theorem states that under suitable conditions a Fourier series can represent any periodic function. Many different applications can be derived from this. Sound waves expressed as amplitude changing over time can be represented as Fourier series.

A function with respect to time is described as belonging to the time domain. The Fourier transform is a mathematical operation that transforms a function expressed in the time domain into the frequency domain. An inverse Fourier transform converts from the frequency domain to the time domain. These transforms deal with infinite sum of periodic signals.

4.2.2 Complex Numbers

A Fourier series is composed of complex coefficients. Each coefficient has two components: Real and Imaginary. The frequency domain represents the spectrum of frequencies present in a given sampled signal. The Real component represents the magnitude of the detected frequency, while the Imaginary component represents the phase offset of the waveform.

4.2.3 The Discrete Fourier Transform

The Discrete Fourier Transform (DFT) deals with discrete time and discrete frequency values. Digital sampling takes a particular value at a given time of a continuous signal and so delivers a Discrete Time Signal. The DFT converts a finite segment of an implied infinitely continuous periodic signal into the frequency components of that segment. Because the DFT assumes that the finite segment is one period of an infinitely repeating set, the amplitude of signals at the edges of the sample might not be in phase. This leads to artefacts in the output. A windowing function is applied to reduce the amount of input signal analysed at the edges of the sampled segment.

4.2.4 Windowing

The DFT treats the input sample buffer as a continuous function. A problem occurs when the wavelength of a detected signal does not coincide symmetrically with the sample rate.

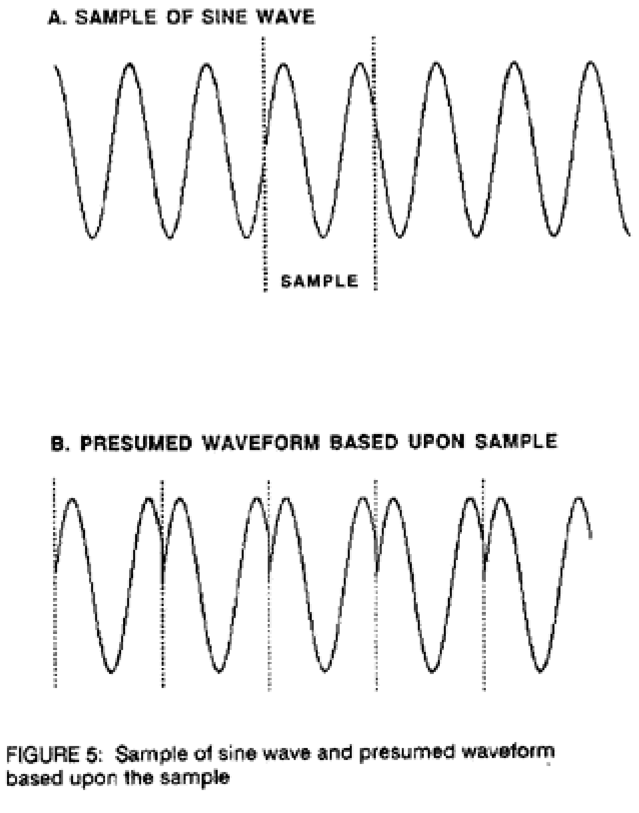

A. SAMPLE OF SINE WAVE
SAMPLE
B. PRESUMED WAVEFORM BASED UPON SAMPLE
FIGURE 5: Sample of sine wave and presumed waveform based upon the sample

Figure 21 – Artefact of continuous DFT function [V] **(Courtney, 2012)**

This can be a cause of aliasing and spectral leakage. To prevent this phenomenon a window function is used to suppress the intensity of the amplitude of the signal at the edges of the buffer.

Many different window functions exist, with different characteristics useful for particular applications. For audio analysis, the two most common examples are the Hamming and Hann (often mistaken as Hanning)

Window Functions. The samples from the input buffer are multiplied with the coefficients of the Window function. This affects the intensity of the corresponding frequency domain output.

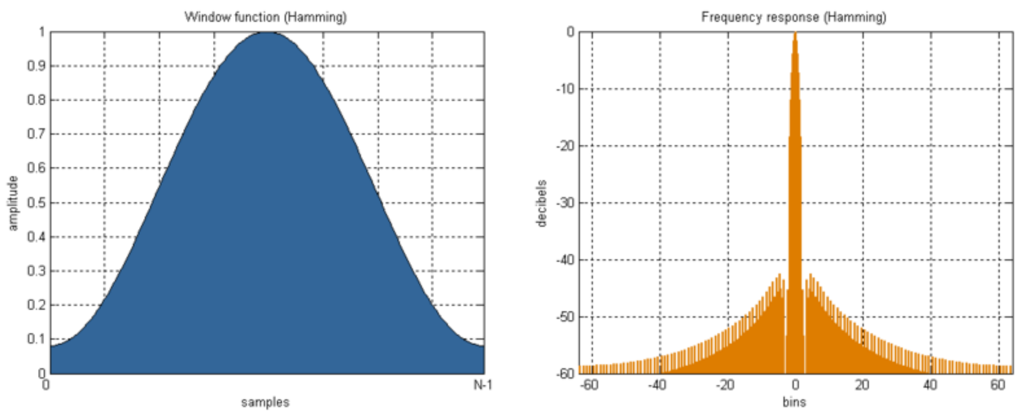


Figure 22 – Hamming Window Function **(Wikipedia, 2005)**

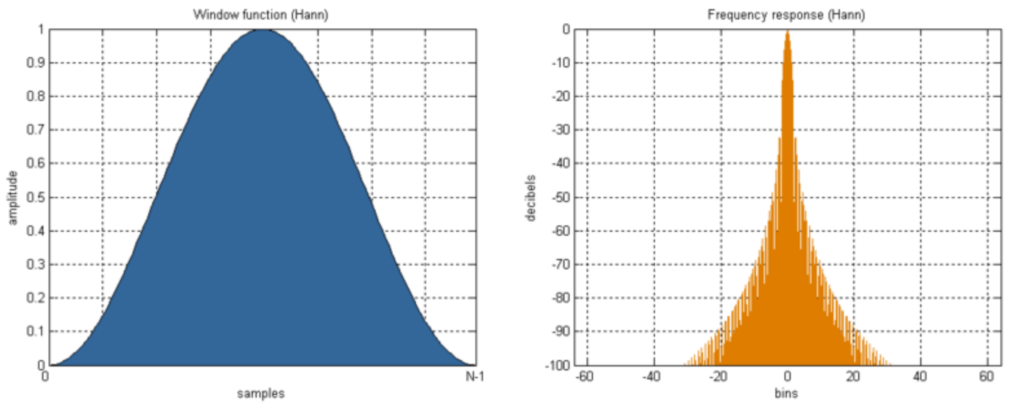


Figure 23 – Hann Window Function **(Wikipedia, 2005)**

4.2.5 The Fast Fourier Transform

The Fast Fourier Transform (FFT) is a computationally efficient algorithm for calculating a Discrete Fourier Transform. There are many different implementations of the FFT, the most famous and popular being the Radix-2 Decimation-In-Time (DIT) Cooley Tukey algorithm. It uses the ‘divide and conquer’ approach to efficiently reduce the number of complex multiplications and additions required for a DFT by splitting the number of samples into pairs (odds and evens) repeatedly. Various code implementations have been devised, using recursion or nested for loops.

The Propeller's REV instruction is useful for rearranging the indexes of the buffer in preparation for the multiplication and addition stage. Many C implementations use for loops to achieve what the Propeller can do with one instruction.

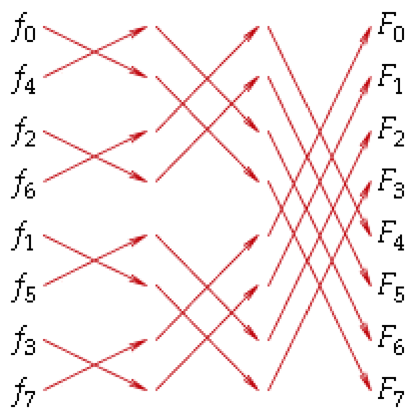


Figure 24 – Butterfly diagram representing decimation of samples where N=8

[VI] **(Chernenko, 2007-2012)**

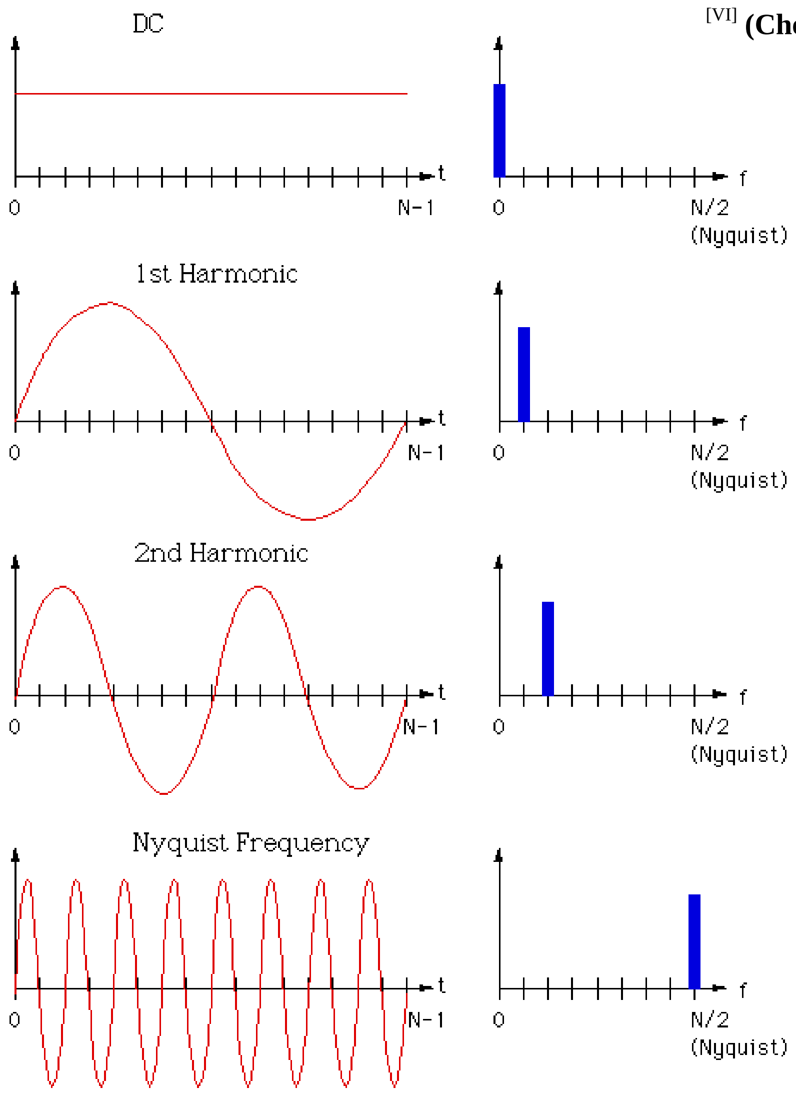


Figure 25 – The relationship between periodic signal component in the time domain and the frequency bins returned by the DFT.

[III] **(Bourke, 1993)**

4.2.6 Sample Rate vs Frequency Resolution

There is a trade off between frequency resolution and time resolution when performing an FFT. The number of samples in the transform is the number of evenly spaced frequency bins that are output. The offset between each bin is dependent on the sample rate and number of samples in the transform.

The highest frequency output from the FFT is half the sampling frequency, or Nyquist Frequency. If an audio signal sampled at 44 KHz is processed using a 1024 sample FFT then the frequency bins will be equally spaced from 0 to the Nyquist Frequency - 1 with resolution of ~21.5 Hz.

$$Frequency\,Bin\,Size = \frac{(Sample\_Rate/2)}{Number\ of\ FFT\ samples}$$

4.2.7 Varieties and Alternatives of FFT Algorithms

Many variations of code and algorithmic implementations of the FFT already exist. There are versions for just about every coding language which have been compared and benchmarked against each other.

An algorithm similar to the FFT is the Fast Hartley Transform. It also uses butterfly loops; however it only uses real values from 2 to N/2, while the FFT loops from 1 to N. This decreases the amount of memory required to perform the transform and the number of loop iterations by about half.

There is an existing implementation of this algorithm on the Parallax forums. [XII] **(ngl, 2009)**

One solution to the problem of the time-frequency resolution trade-off is the Short Time Fourier Transform. This uses a sliding sampling window (using half of a previous buffer of inputs) to increase accuracy in the time domain. Since each processed sample contains duplicate information, it is possible to determine more accurately the moment of attack of a note.

Frequencies detected within the range of a bin will increase corresponding magnitude in proportion to the signals intensity. When a frequency does not sit exactly on the edge between two bins, its intensity shall be split between the two bins. To determine the actual frequency detected between two or more bins, a technique called quadratic interpolation is used.

One solution to this problem is the Constant Q transform. This uses the FFT with logarithmically spaced sample input. This results in frequency bins which are also logarithmically spaced. The human perception of musical notes is logarithmically based, which means each note falls within its own frequency bin and there is no need for quadratic interpolation to determine which note has been detected.

A demonstration of the Constant Q transform on the Propeller is available on YouTube and also referenced to at the end of a website describing the transform. [VII] **(Digparty, 2012)**

4.3 Parallax FFT Discussions

During research for information on FFTs I discovered a Wiki entry on the Parallax website showing a fixed point FFT implementation that has been translated from C to Propeller Assembler and then hand optimised. The small amount of comments buried in the code stated the object would output to a 1bpp 320x240 bitmap. It seemed too good to be true. The code was simply a PASM section that would run in a cog, so I set about building the necessary structures around it to be able to run it. The Propeller library VGA example object uses a 512x384 bitmap. Luckily someone has uploaded the modified version to the object exchange.

The next challenge was deciding how to interface the tiles of the screen to the plot subroutine. The pointers for the screen, real and imaginary buffers were hard coded as Hub RAM locations. This led to two possible courses of action. Either assign specific locations of Hub RAM for the buffers or replace the hard coded values with dynamically assigned pointers passed from calling method.

The latter allows instantiation of the same object multiple times, passing Hub RAM pointers to locations allocated by the calling object at runtime.

The Propeller forum has a large number of enthusiastic members who are discussing how best to help others understand the inner workings of the FFT. A thread has been started titled “Fourier for Dummies” which aims to write a how-to guide for people curious for an understanding of the algorithm.

http://forums.parallax.com/showthread.php?127306-Fourier-for-dummies-under-construction/

[II] **(Bean, 2010)**

The Propeller has several optimized versions of the FFT, including use of the Sine and Log tables in ROM for faster calculations. Object encapsulation has been implemented by using a mailbox delivery system. The object offers a flag register and a mailbox register to the calling object. These are used to ‘post’ an item to be processed. The content of RAM pointed at from the mailbox register contains the completed FFT output when the flag is cleared.

5 Frequency Counter Implementation

The attack of a musical note can begin at any time. Therefore, a method for determining if the given 512 sample window contains a complete waveform is required.

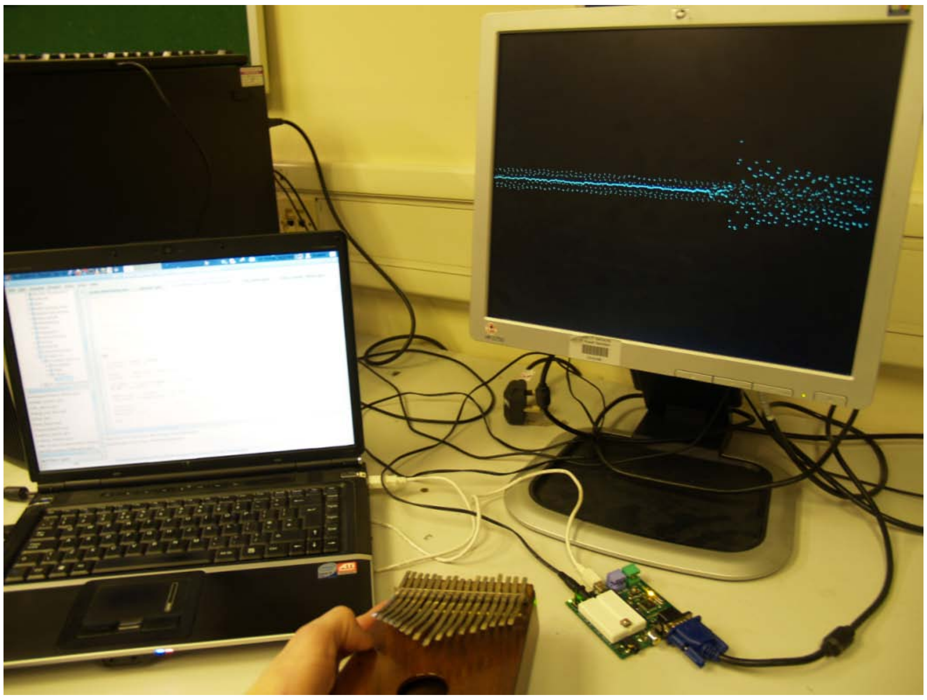
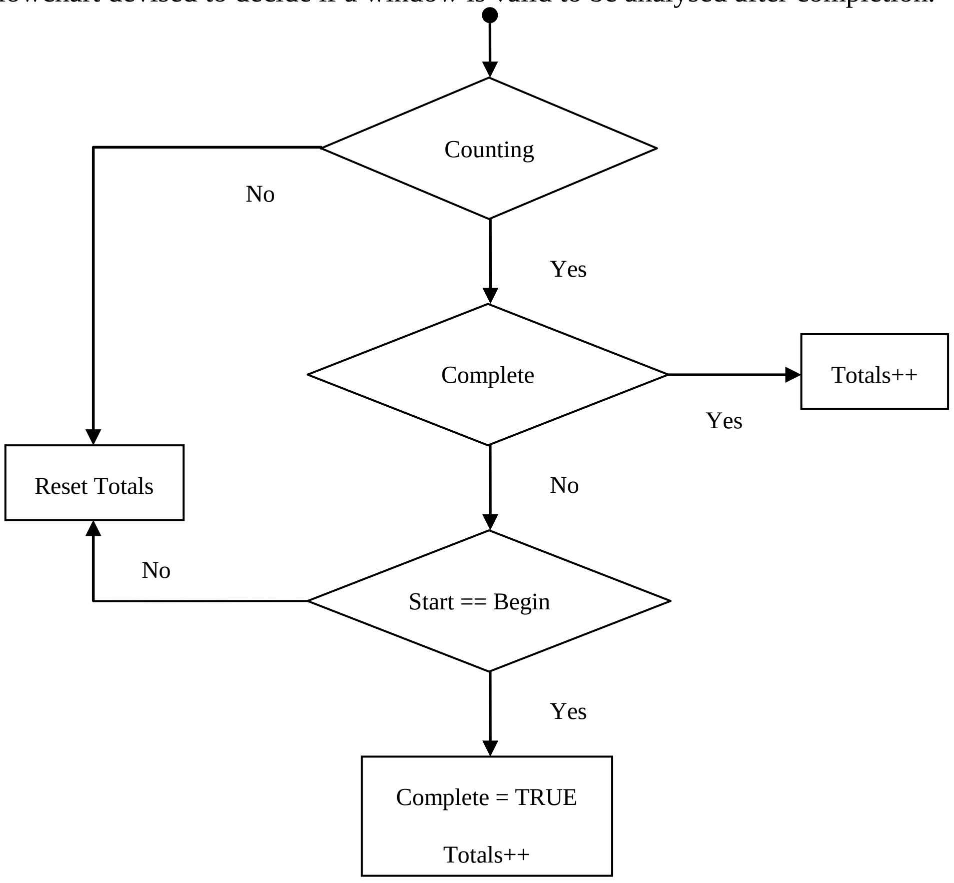


Figure 26 – Example of Attack detected in the middle of a sample window

Below is the flowchart devised to decide if a window is valid to be analysed after completion.



The solution involves checking if the current sample is above the preset threshold. The variables used are: counting and complete flags, window begin and signal start time clock values and a cycles counter. The value of clock register is saved at the beginning of each new window. When a new signal that above a preset threshold is detected the value of the clock register is saved. The counting flag is set to true. At the end of each pass of the buffer, if counting is true then the start time is compared with the begin time. Only if they match, then there is a complete window of samples and a valid value of zero crossings is read from the cycles count variable. Start and Begin variables are then set to the current clock value and the process begins again. When the signal drops below the preset threshold, the counting and complete flags are set to false.

The phase of the signal will most likely not coincide with the edges of the window. This means that number of zero crossing of a standard wave may vary between each 512 sample window. After a window has been analysed, its number of zero crossings is added to a running average.

The choice of averaging technique plays a large role in the accuracy of the detected frequency. The frequency of a wave may vary between the moment of attack and the moment its amplitude becomes undetectable, due to energy lost during vibration. If every window’s running total is added, then the result will be an inaccurate representation of the note’s frequency at the moment of attack, which is when its tuning is most noticeable.

Because of this, the chosen implementation displays the current frequency estimation as each window is analysed, and when the signal has fallen below the detectable threshold the total average is displayed and remains on the screen. This allows changes of estimation over time to be observed.

The highest note on the Kalimba is ~1250 Hz so the chosen sample rate is between 4 KHz and 7 KHz, with a 1500 Hz low pass filter.

One problem is that if there is a background noise that doesn’t allow the detected signal to fall below the threshold before the note has finished. The total average of frequency estimation changes to encompass the background noise and accuracy is lost. To resolve this, the tuner must be used in a quiet environment and a note should be played as cleanly as possible. This will achieve maximum accuracy in reading.

Below is a recorded waveform of a Kalimba playing the note A4.

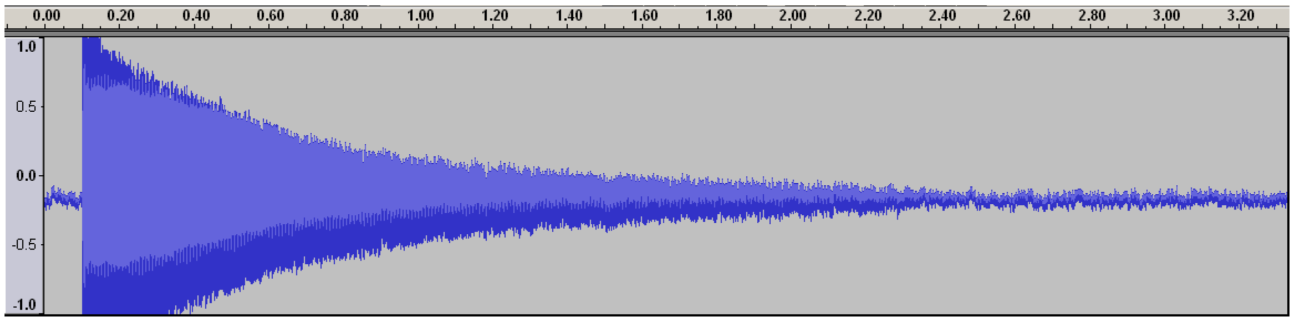


Figure 27 – Kalimba Playing A4

The attack of the note and the subsequent reduction in amplitude can clearly be seen. The next image shows magnification of the same waveform.

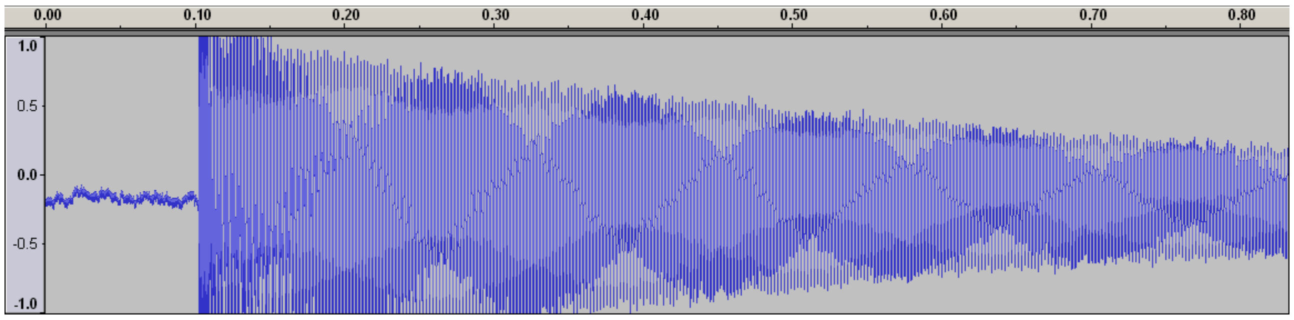


Figure 28 – Magnification of Kalimba Playing A4

The superposition of waveforms can be seen. These represent the interaction of the various overtones and harmonics produced by the resonance of the Kalimba.

The pictures below show the same superposition of waves just after the attack of the note using the Microphone to VGA object running on the Propeller.

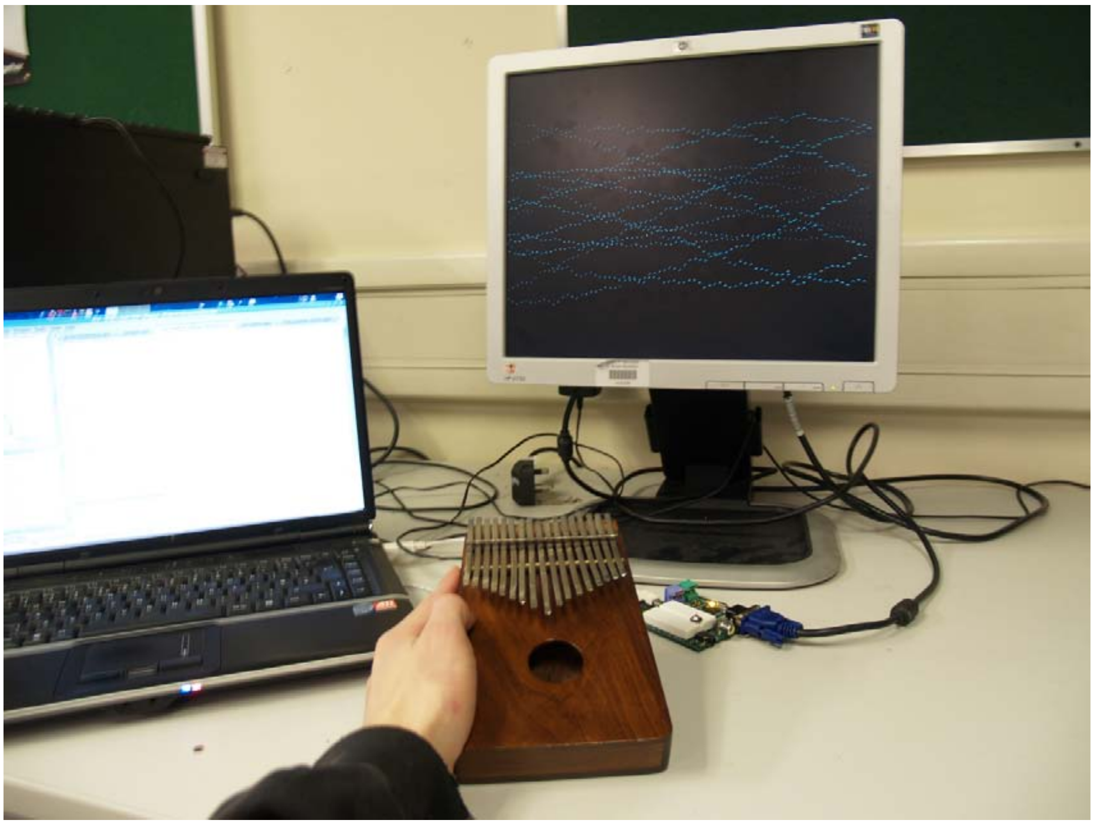


Figure 29 – Interference during attack – Example 1

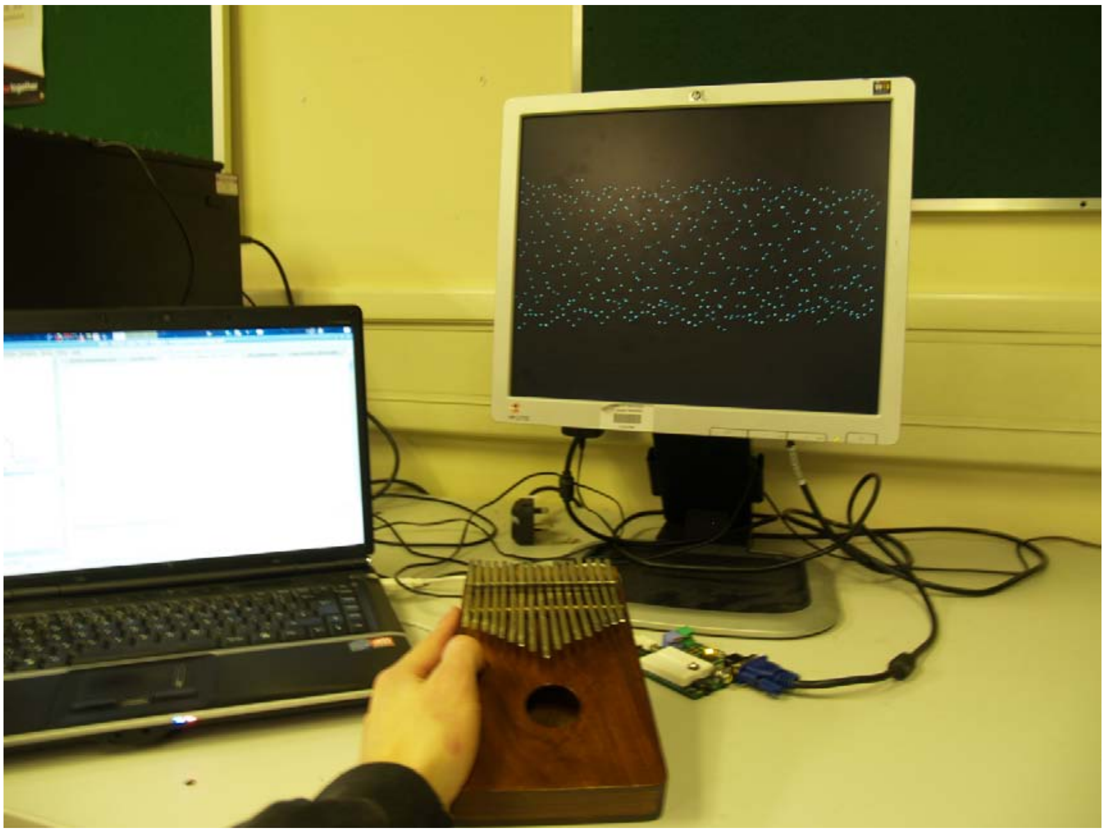


Figure 30 – Interference during attack – Example 2

The overtones produced will distort the wave, but will not affect the number of zero crossing of the fundamental note. Further magnification demonstrates this.

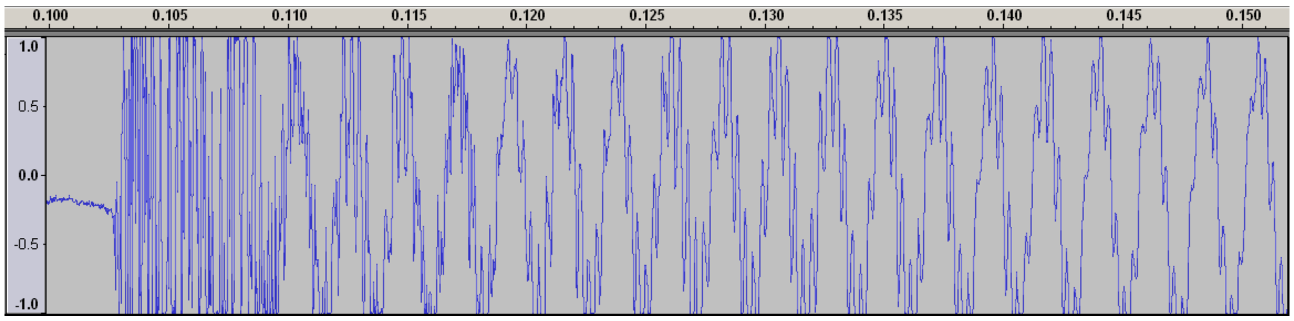


Figure 31 – Further Magnification of Kalimba Playing A4

One of the most time consuming problems encountered with this project was correct extrapolation of notes from frequency input. Testing methods included: exporting intermediate values to the serial console, using existing code examples to double check input and output values, using Excel to verify expected code formulae behaviour. This problem needs further investigation to complete the application

The frequency counter readings appear to be accurate, the note and cent readings, while stable, are not accurate. The note shown is normally in the range of a few notes of the note expected.

Here are some screen shots depicting the finished user interface.

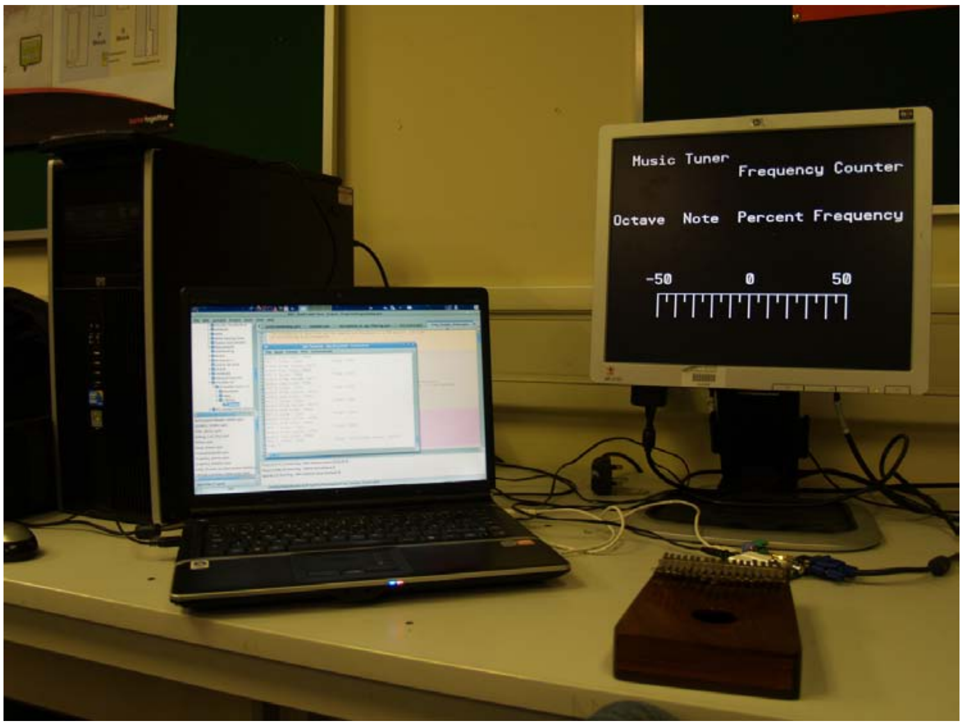


Figure 32 – Frequency Counter Graphical User Interface Demo

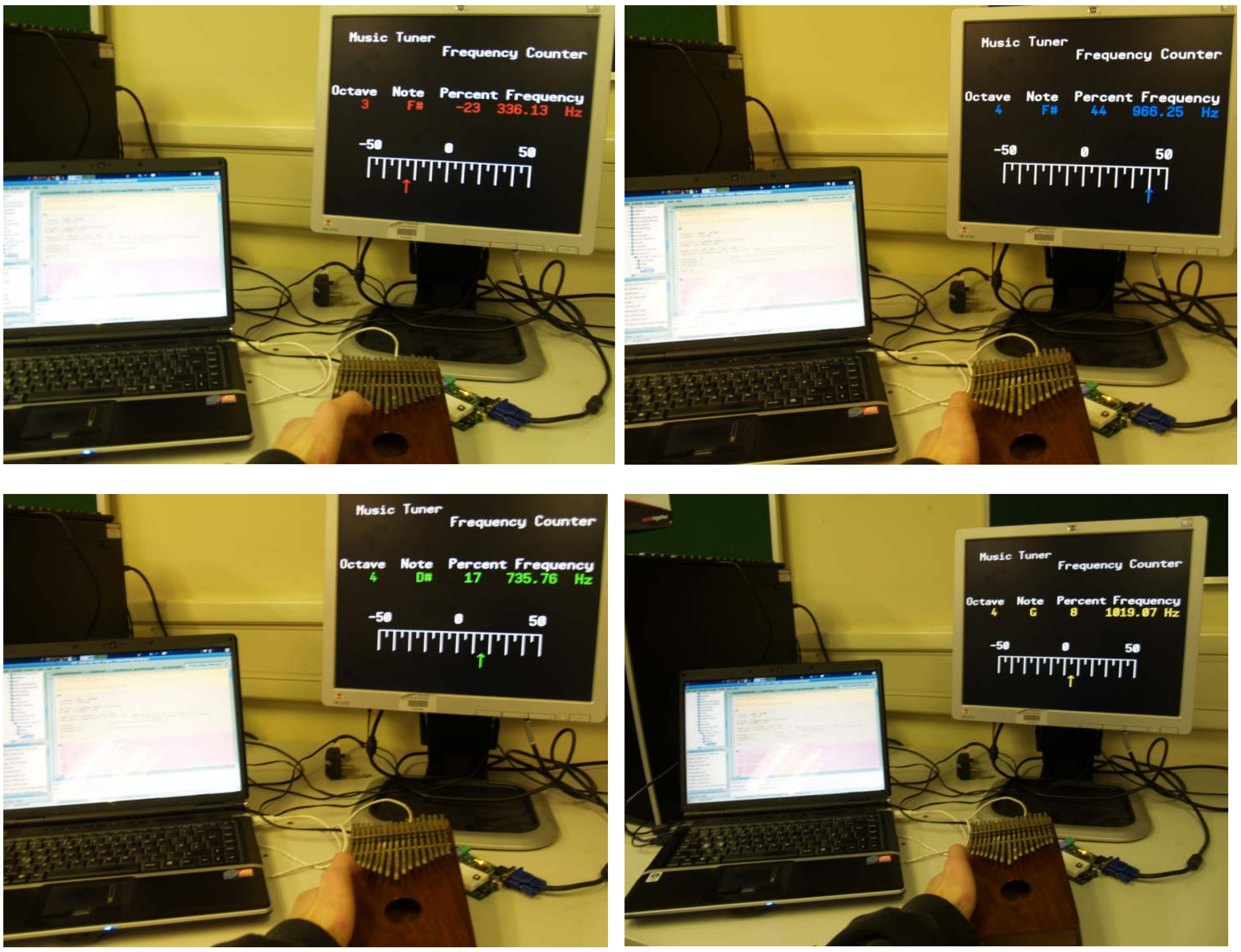
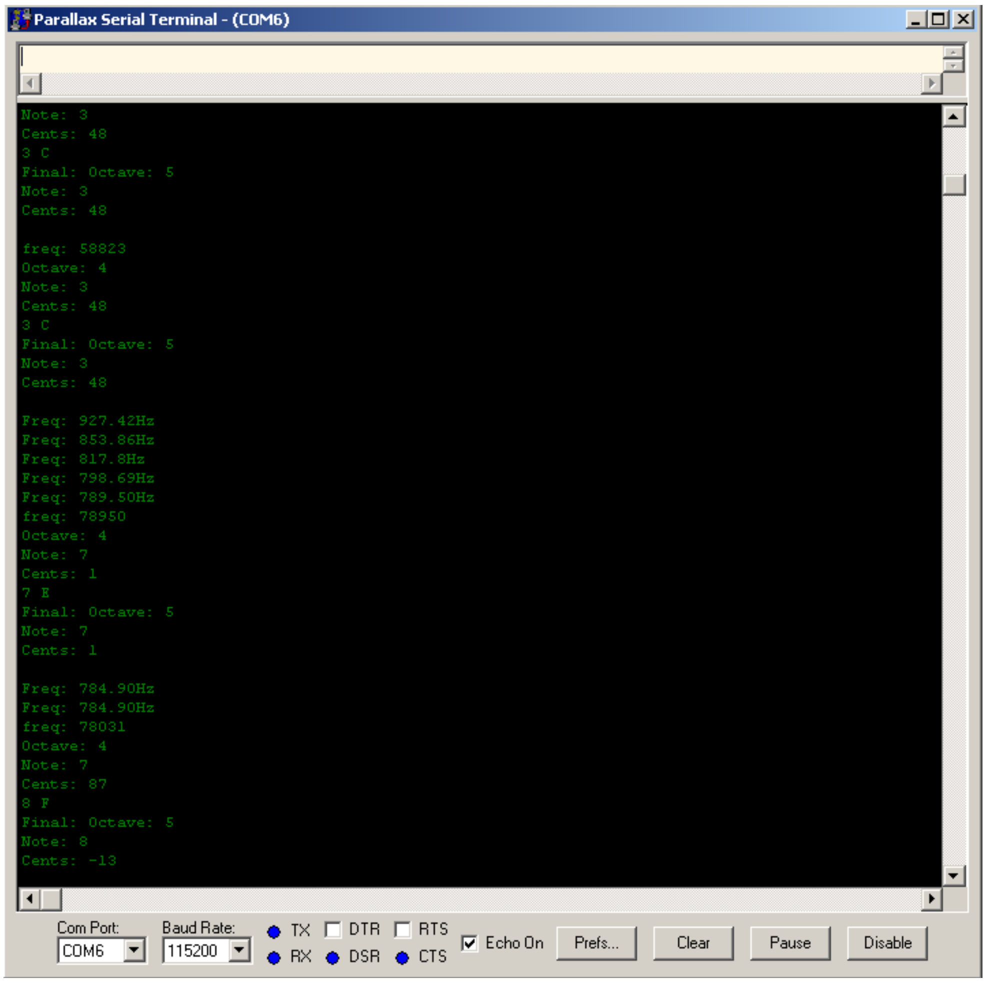


Figure 33 – Frequency Counter In Action



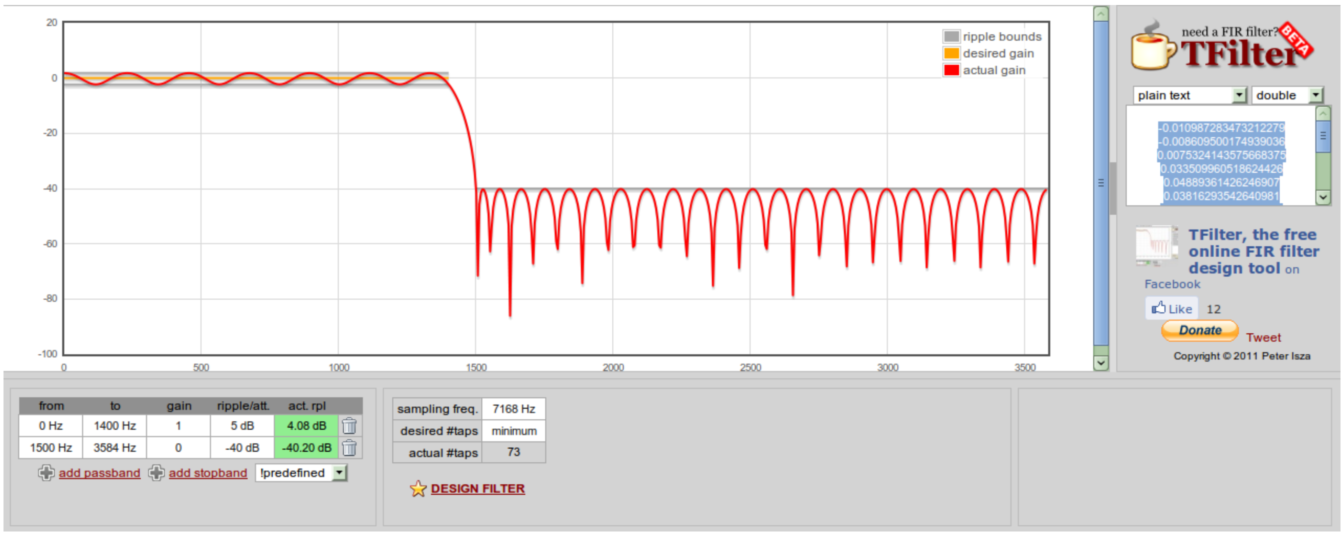
```
Note: 3
Cents: 48
3 C
Final: Octave: 5
Note: 3
Cents: 48

freq: 58823
Octave: 4
Note: 3
Cents: 48
3 C
Final: Octave: 5
Note: 3
Cents: 48

Freq: 927.42Hz
Freq: 853.86Hz
Freq: 817.8Hz
Freq: 798.69Hz
Freq: 789.50Hz
freq: 78950
Octave: 4
Note: 7
Cents: 1
7 E
Final: Octave: 5
Note: 7
Cents: 1

Freq: 784.90Hz
Freq: 784.90Hz
freq: 78031
Octave: 4
Note: 7
Cents: 87
8 F
Final: Octave: 5
Note: 8
Cents: -13
```

Figure 34 – Example Terminal Output for Frequency Counter Testing

Parallax forum regular, Phil Pilgrim (PhiPi), has provided an online service for converting FIR filter parameters to PASM code for real-time audio filtering. The instructions and service are documented on his forum post.

[XXI] **(Pilgrim, 2011)**

Four FIR low-pass filters with a 1.5 KHz cut off were generated for sampling at 4, 5, 6 and 7 KHz. These were included in the frequency counter and spectrograph projects.



```
-0.010987283473212279
-0.008609500174939036
0.007532414357566837
0.03350996051862442
0.048893614262469
0.038162935426409
```

from	to	gain	ripple/att.	act. rpl
0 Hz	1400 Hz	1	5 dB	4.08 dB
1500 Hz	3584 Hz	0	-40 dB	-40.20 dB

sampling freq.	7168 Hz
desired #taps	minimum
actual #taps	73

Figure 35 – Low Pass Filter with 1500 Hz cut-off

6 Spectrogram Implementation

Spectrogram generation is achieved using a similar method to the Microphone to VGA object. However, instead of the display updating from left to right as the signal changes, each index of the X axis represents a frequency range and position in the Y axis represents intensity of the detected frequency.

To determine the intensity of component frequencies in a given signal, the Fast Fourier Transform will be used.

Microphone input is sampled and temporarily stored in cog RAM. After the maximum number of samples has been recorded, the sampling object checks the status flags of available FFT objects. When an available FFT is detected, the sample buffer is moved the relevant FFT's input buffer in Hub RAM and the flag is changed to indicate to the object that it has data to work on.

The implementation chosen allows the dynamic creation of between 1 and 4 FFT objects to work on sampled data. The reason for this is that an FFT operation can take longer than the sampler object takes to fill its buffer. This is dependent on the sample rate chosen. Based on test output, 72ms is required for a 1024 sample FFT.

Once an FFT object has finished operating on its input buffer, the output is then analysed and plotted on VGA pixel array by XORing the relevant pixels. The advantage to this approach is that multiple FFT's can update display their output when they are ready, since the VGA array is not a mutually exclusive resource. This allows the user to see spikes of particular frequency ranges appear in a timely manner.

The disadvantage to this approach is that the display becomes littered with previous output. The solution to this problem is to periodically clear the display.

One of the challenges encounter was a mechanism to pass variable numbers of FFT buffer and flag pointers to the sampler object. Initial attempts involved commenting out relevant sections of source code. This seemed inelegant and further solutions were investigated.

Propeller Assembler supports self-modifying code. The source or destinations fields of an instruction register can be manipulated using the `MOVS` and `MOVD` instructions. The instruction field may also be modified with the `MOVI` instruction.

Using a variable passed to the sampler cog, the number of FFT objects initialised can be determined dynamically and the appropriate locations in cog RAM are populated accordingly.

6.1 Design

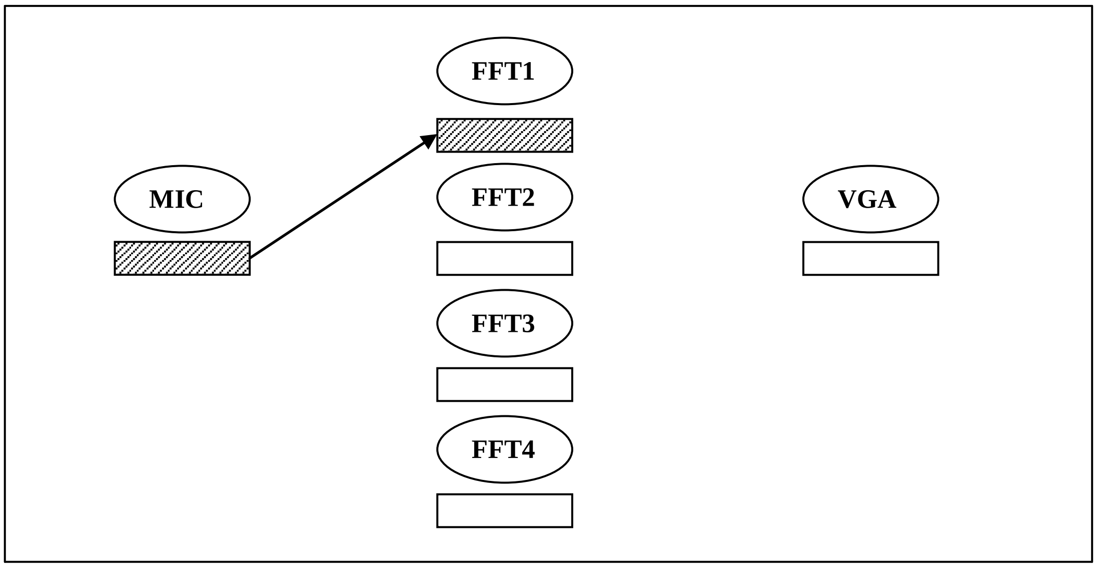
This is the chosen design for the spectrograph. Since FFT calculations can take longer than the time it takes to complete a sample buffer, this design approach utilises the multi-core architecture to perform multiple FFT operations at once and produce seamless graphical output.

Due to the restrictions of Hub RAM size and the number of cogs available, a maximum of four FFT objects can be instantiated at once.

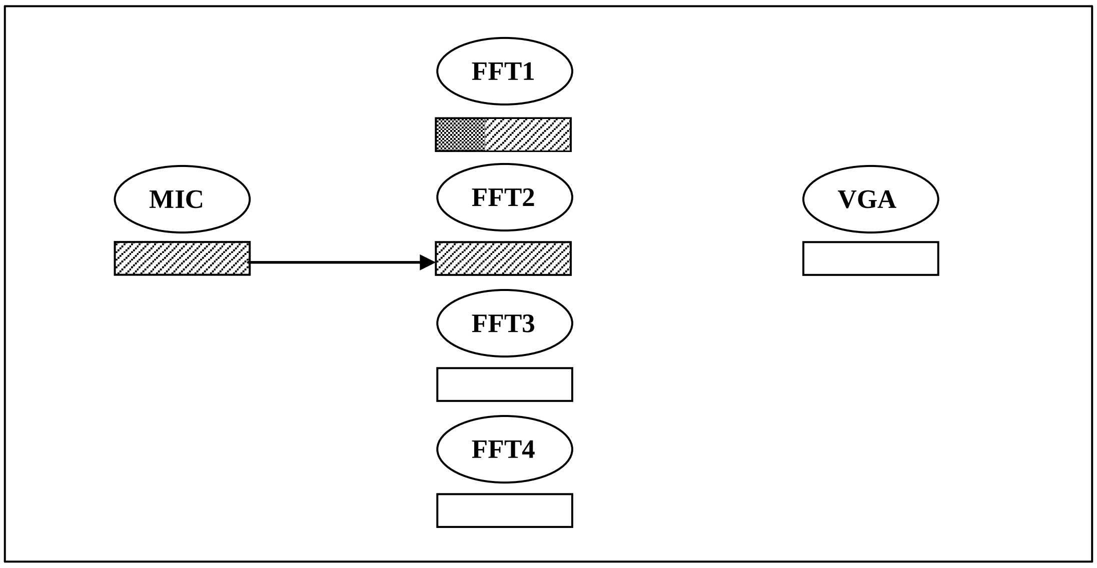
1024 microphone samples are taken and then passed to Hub RAM, into the input buffer of the first available FFT object. When the FFT has finished it plots the intensities of the component frequencies to the screen.

From left to right, each pixel horizontally represents one frequency bin. The intensity of the frequency is represented by the distance of the pixel plotted from the bottom of the screen. One pixel per horizontal line is XOR-ed. Each FFT has access to the screen buffer and the plots become superimposed on top of each other.

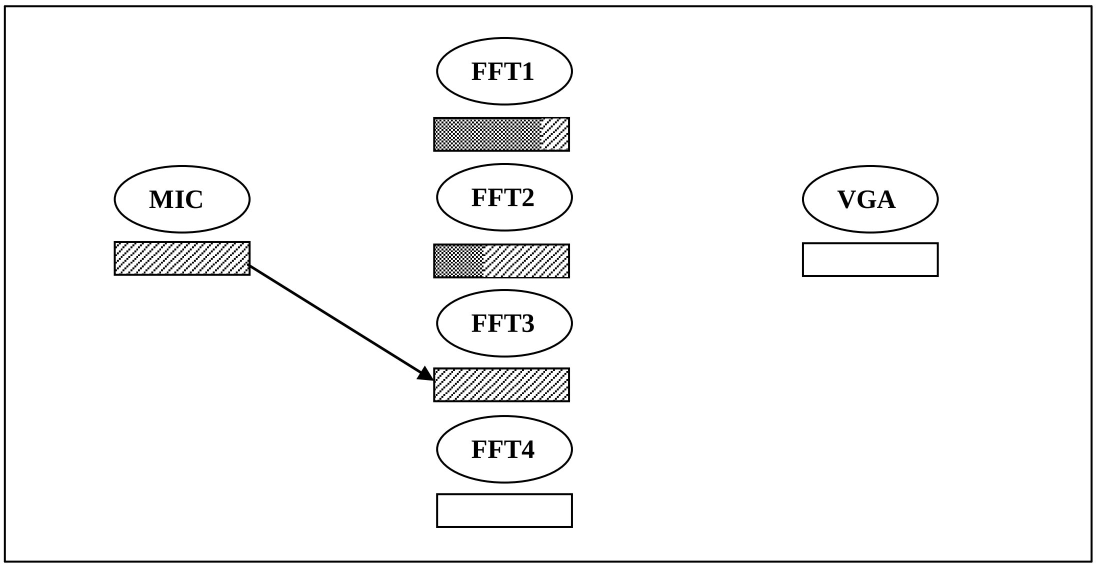
The following sequence of diagrams depicts how flows throughout the system.



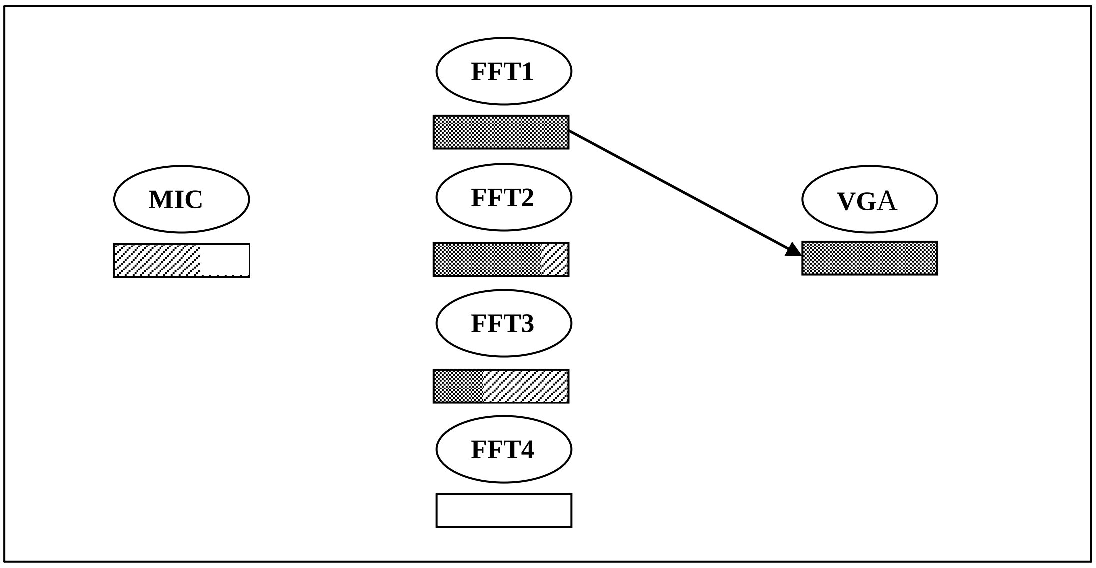
The diagram above shows the microphone buffer being copied to the first FFT buffer for processing.



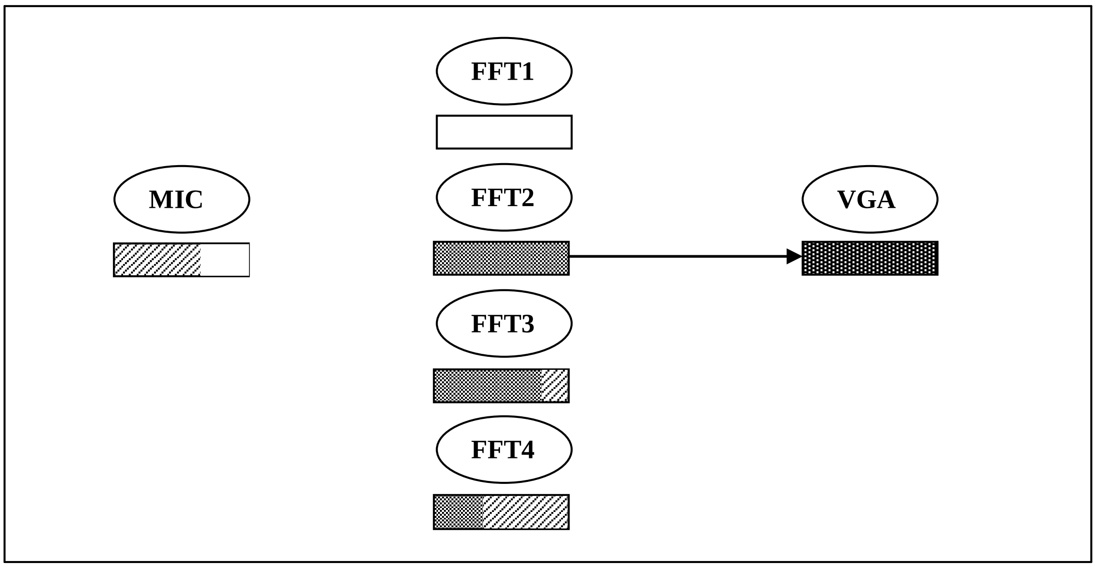
The diagram above shows FFT1 processing the data while the microphone object continues to sample. Once the microphone sample buffer is full again, the data is copied to the input buffer of FFT2.



The diagram above shows FFT1 nearing completion as the next sample buffer is copied to FFT3.



The diagram above shows the complete FFT1 plotting the component frequencies of its samples to the VGA pixel buffer. The FFT1 buffer then becomes available to the sampler again.



The diagram above shows the complete FFT2 plotting the component frequencies of its samples to the VGA pixel buffer. In the diagram above the same process has continued and FFT4 has been filled and triggered.

The method for one FFT is slightly different. The Sampler object stalls writing to the buffer until the FFT is available. This creates visual feedback on the spectrogram of another spectrogram offset from the original. This is used as a visual indication that stall has occurred, together with serial output.

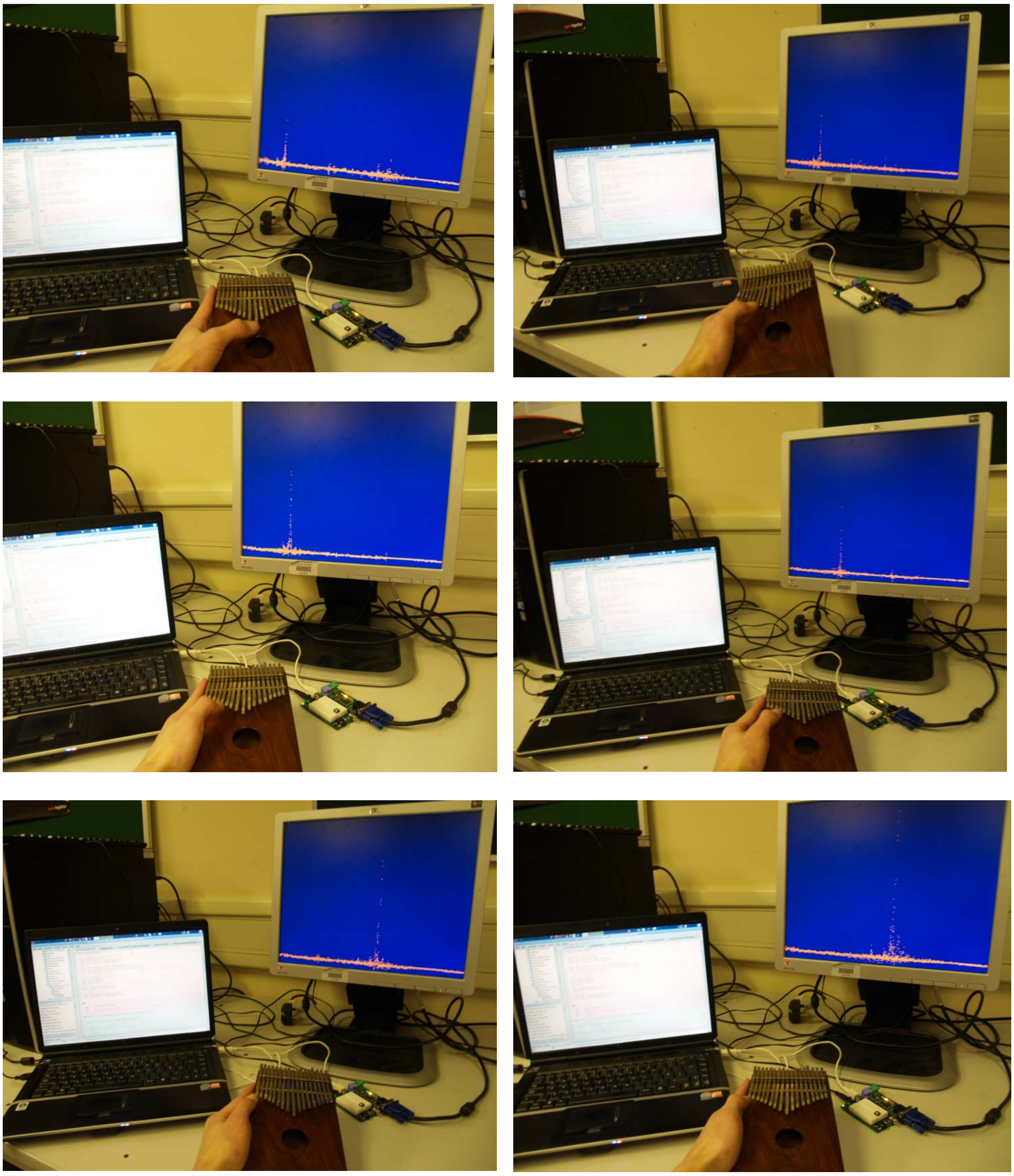


Figure 36 – Spectrograph In Action

6.2 Testing Variable Number of FFTs at Different Sample Rates

Performance tests were carried out to measure the capabilities of the system using multiple FFT objects processing various sized buffers at various sample rates.

Results from the testing phase are below. Full test results are in the Appendix.

FFT Sample Size	Number of FFTs	Max Sample Rate (KHz)	RAM Usage (Longs)			RAM Usage (Percentage)		
			Program	Variable	Free	Program	Variable	Free
512	1	14	1,450	3,072	3,666	17.71	37.52	44.77
	2	14	1,451	3,592	3,145	17.72	43.87	38.41
	3	28	1,452	4,112	2,624	17.73	50.22	32.05
	4	42	1,452	4,632	2,104	17.73	56.57	25.70

FFT Sample Size	Number of FFTs	Max Sample Rate (KHz)	RAM Usage (Longs)			RAM Usage (Percentage)		
			Program	Variable	Free	Program	Variable	Free
1,024	1	14	1,450	3,584	3,154	17.71	43.77	38.52
	2	14	1,451	4,616	2,121	17.72	56.38	25.90
	3	28	1,452	5,648	1,088	17.73	68.98	13.29
	4	42	1,452	6,680	56	17.73	81.58	0.68

FFT Sample Size	Number of FFTs	Max Sample Rate (KHz)	RAM Usage (Longs)			RAM Usage (Percentage)		
			Program	Variable	Free	Program	Variable	Free
2,048	1	10	1,450	4,608	2,130	17.71	56.28	26.01
	2	13	1,452	6,664	72	17.73	81.39	0.88

7 Conclusions

This project has given me a much deeper understanding of the complexities involved with audio analysis and Digital Signal Processing. The Propeller is an excellent platform for learning to implement sophisticated algorithms in a multi-processor environment. My experiences throughout this project have sparked other interesting Propeller project ideas for the future.

I feel happy with the results of the graphical tuner. Some issues still exist in determining the correct note and percentage offset values; however I feel that the requirements were met and I am confident that the remaining problems will be readily solved.

I am also very pleased with implementation of the FFT powered spectrograph. The challenges involved with chaining the sampler object to various FFT objects has brought my knowledge of Propeller Assembler up a level. The research undertaken into FFTs on the Propeller encouraged me to communicate with other software engineers who share an interest in this field. The Propeller community is thriving with enthusiastic and friendly people that are willing to help and share knowledge. Since becoming a member of the open source community I have embraced the values it upholds. All of my final work shall be freely available for others to look at, learn from, modify and use in their own projects.

The design chosen for multiple FFTs concurrently would easily suit implementation of the Short Time Fourier Transform for high frequency domain and time domain resolution.

Multiple note representation requires TV object output. Due to restricted access to required hardware that I do not own at present, notably a TV, this portion of the project aims has yet to be completed. However I intend to make progress in this area before demonstration.

An extension to the theme of analogue to digital audio analysis is digital to analogue audio generation.

Possible extensions could include:

Tuning forks are a traditional method of producing a note in its ‘true’ form. Using the audio generating abilities of the Propeller Demo Board, as well as its keyboard and mouse inputs, it would be possible to create an interactive digital tuning fork with the full range of notes available.

Metronomes are for keeping in time whilst playing an instrument. They produce audible clicks at a specific rate. This would be very easy to implement and could be a good introductory challenge for programming the Propeller.

Music Minus One, where other instruments play the backing for a musician to play along with visual prompting.

I think the Propeller has the potential to fulfil all of these applications, and many more. I would encourage its use in the academic environment.

8 References

[I]. Bean (2010) *Fourier for dummies - under construction - Parallax Forum*, 29 Nov, [Online], Available: http://forums.parallax.com/showthread.php?127306-Fourier-for-dummies-under-construction&p=957531&viewfull=1#post957531 [03 Feb 2012].

[II]. Bourke, P. (1993) *Fast Fourier Transform*, [Online], Available: http://paulbourke.net/miscellaneous/dft/ [05 Jan 2012].

[III]. Campbell, B. (2009) *bst features*, 07 Mar, [Online], Available: http://www.fnarfbargle.com/features.html [12 Mar 2012].

[IV]. Courtney, D. (2012) *Introduction to Spectrum Analysis*, [Online], Available: http://chandrakantha.com/articles/spectrum/spectrum.html [06 Jan 2012].

[V]. Chernenko, S. (2007-2012) *Fast Fourier Transform - FFT - librow*, [Online], Available: http://www.librow.com/articles/article-10 [01 Mar 2012].

[VI]. Digparty (2012) *Learn and talk about Constant Q transform, Harmonic analysis, Integral transforms*, [Online], Available: http://www.digplanet.com/wiki/Constant_Q_transform [01 Feb 2012].

[VII]. Encyclopædia Britannica Online (2012) *Fourier series*, [Online], Available: http://www.britannica.com/EBchecked/topic/215122/Fourier-series [15 February 2012].

[VIII]. Holdaway, M. (2006-2012) *Kalimba Magic*, [Online], Available: http://www.kalimbamagic.com/alto [25 Nov 2011].

[IX]. Humanoido (2009) *Parallax Forum - ULTIMATE-List-of-Propeller-Languages*, 22 May, [Online], Available: http://forums.parallax.com/showthread.php?113091-ULTIMATE-List-of-Propeller-Languages [20 February 2012].

[X]. Ingle, A. (2011) *The Modified Constant Q Spectrogram and its Application to Phase*, 13 May, [Online], Available: minds.wisconsin.edu/bitstream/handle/1793/53713/atul_ingle_MS_thesis.pdf [20 Feb 2012].

[XI]. ngl (2009) *Fast Hartley Transform - Parallax Forum*, 30 Oct, [Online], Available: http://forums.parallax.com/showthread.php?117202-Fast-Hartley-Transform [02 Feb 2012].

[XII]. Parallax Inc. (2006) *Propeller Manual v1.01*, Rocklin: Parallax Inc.

[XIII]. Parallax Inc. (2006-2011) *Propeller Manual v 1.2*, Rocklin: Parallax Inc.

[XIV]. Parallax Inc. (2007) *Propeller Demo Board*, 12 December, [Online], Available: http://www.parallax.com/Store/Microcontrollers/PropellerDevelopmentBoards/tabid/514/CategoryID/73/List/0/SortField/0/catpageindex/2/Level/a/ProductID/340/Default.aspx [20 February 2012].

[XV]. Parallax Inc. (2007) *Propeller Downloads*, 26 November, [Online], Available: http://www.parallax.com/ProductInfo/Microcontrollers/PropellerGeneralInformation/PropellerDownloads/tabid/832/Default.aspx [20 February 2012].

[XVI]. Parallax Inc. (2010) *2009/10 Propeller Contest*, [Online], Available: http://www.parallax.com/tabid/846/Default.aspx [12 Mar 2012].

[XVII]. Parallax Inc. (2011) *Propeller Help Version 1.2*, 17 June, [Online], Available: 'C:/Program Files/Parallax Inc/Propeller Tool v1.3/Help/Content/SpinTutorTopics/SpinLesson01/Spin1-Objects.htm' [19 February 2012].

[XVIII]. Parallax Inc. (2011) *Propeller P8X32A Datasheet v1.4*, Rocklin: Parallax Inc.

[XIX]. Parallax Inc. (2011) *QuickStart 1: Comparison of Programming Tools*, [Online], Available: http://www.parallaxsemiconductor.com/quickstart1 [04 Feb 2012].

[XX]. Pilgrim, P. (2011) *FIR2PASM: Automatic FIR Filter Code Generator*, 20 Jul, [Online], Available: http://forums.parallax.com/showthread.php?133173-FIR2PASM-Automatic-FIR-Filter-Code-Generator&highlight=lowpass [03 Jan 2012].

[XXI]. Smith, J.O. (2007) *Mathematics of the Discrete Fourier Transform (DFT)*, [Online], Available: https://ccrma.stanford.edu/~jos/mdft/Alias_Operator.html [01 Mar 2012].

[XXII]. William Collins Sons & Co. Ltd. (2009) *Collins English Dictionary - Complete & Unabridged 10th Edition*, London: HarperCollins Publishers.

9 Bibliography

Ackers, F. (2004) *FFT of waveIn audio signals - CodeProject®*, [Online], Available: http://www.codeproject.com/Articles/6855/FFT-of-waveIn-audio-signals [26 Nov 2011].

Barr Group (2012) *Introduction to FIR Digital Filter*, [Online], Available: http://www.barrgroup.com/Embedded-Systems/How-To/Digital-Filters-FIR-IIR [15 Jan 2012].

Bean (2010) *Fourier for dummies - under construction - Parallax Forum*, 29 Nov, [Online], Available: http://forums.parallax.com/showthread.php?127306-Fourier-for-dummies-under-construction&p=957531&viewfull=1#post957531 [03 Feb 2012].

Bourke, P. (1993) *Fast Fourier Transform*, [Online], Available: http://paulbourke.net/miscellaneous/dft/ [05 Jan 2012].

Campbell, B. (2009) *bst features*, 07 Mar, [Online], Available: http://www.fnarfbargle.com/features.html [12 Mar 2012].

Chernenko, S. (2007-2012) *Fast Fourier Transform - FFT - librow*, [Online], Available: http://www.librow.com/articles/article-10 [01 Mar 2012].

Courtney, D. (2012) *Introduction to Spectrum Analysis*, [Online], Available: http://chandrakantha.com/articles/spectrum/spectrum.html [06 Jan 2012].

Digparty (2012) *Learn and talk about Constant Q transform, Harmonic analysis, Integral transforms*, [Online], Available: http://www.digplanet.com/wiki/Constant_Q_transform [01 Feb 2012].

Encyclopædia Britannica Online (2012) *Fourier series*, [Online], Available: http://www.britannica.com/EBchecked/topic/215122/Fourier-series [15 February 2012].

Holdaway, M. (2006-2012) *Kalimba Magic*, [Online], Available: http://www.kalimbamagic.com/alto [25 Nov 2011].

Humanoido (2009) *Parallax Forum - ULTIMATE-List-of-Propeller-Languages*, 22 May, [Online], Available: http://forums.parallax.com/showthread.php?113091-ULTIMATE-List-of-Propeller-Languages [20 February 2012].

Ingle, A. (2011) *The Modified Constant Q Spectrogram and its Application to Phase*, 13 May, [Online], Available: minds.wisconsin.edu/bitstream/handle/1793/53713/atul_ingle_MS_thesis.pdf [20 Feb 2012].

Jazzwise Publications (2011) *Seventh String*, 07 Mar, [Online], Available: http://www.seventhstring.com/resources/notefrequencies.html [20 Nov 2011].

ngl (2009) *Fast Hartley Transform - Parallax Forum*, 30 Oct, [Online], Available: http://forums.parallax.com/showthread.php?117202-Fast-Hartley-Transform [02 Feb 2012].

Parallax Inc. (2006) *Propeller Manual v1.01*, Rocklin: Parallax Inc.

Parallax Inc. (2006-2011) *Propeller Manual v 1.2*, Rocklin: Parallax Inc.

Parallax Inc. (2007) *Propeller Demo Board*, 12 December, [Online], Available: http://www.parallax.com/Store/Microcontrollers/PropellerDevelopmentBoards/tabid/514/CategoryID/73/List/0/SortField/0/catpageindex/2/Level/a/ProductID/340/Default.aspx [20 February 2012].

Parallax Inc. (2007) *Propeller Downloads*, 26 November, [Online], Available: http://www.parallax.com/ProductInfo/Microcontrollers/PropellerGeneralInformation/PropellerDownloads/tabid/832/Default.aspx [20 February 2012].

Parallax Inc. (2010) *2009/10 Propeller Contest*, [Online], Available: http://www.parallax.com/tabid/846/Default.aspx [12 Mar 2012].

Parallax Inc. (2010) *Propeller Assembly Language*, 30 Sep, [Online], Available: http://www.parallax.com/portals/0/propellerqna/Content/QnaTopics/QnaAssembly.htm [20 Mar 2012].

Parallax Inc. (2011) *Propeller Help Version 1.2*, 17 June, [Online], Available: C:/Program Files/Parallax Inc/Propeller Tool v1.3/Help/Content/SpinTutorTopics/SpinLesson01/Spin1-Objects.htm [19 February 2012].

Parallax Inc. (2011) *Propeller P8X32A Datasheet v1.4*, Rocklin: Parallax Inc.

Parallax Inc. (2011) *QuickStart 1: Comparison of Programming Tools*, [Online], Available: http://www.parallaxsemiconductor.com/quickstart1 [04 Feb 2012].

Pilgrim, P. (2011) *FIR2PASM: Automatic FIR Filter Code Generator*, 20 Jul, [Online], Available: http://forums.parallax.com/showthread.php?133173-FIR2PASM-Automatic-FIR-Filter-Code-Generator&highlight=lowpass [03 Jan 2012].

Smith, J.O. (2007) *Mathematics of the Discrete Fourier Transform (DFT)*, [Online], Available: https://ccrma.stanford.edu/~jos/mdft/Alias_Operator.html [01 Mar 2012].

Smith, S.W. (2011) *How the FFT works*, [Online], Available: http://www.dspguide.com/ch12/2.htm [29 Dec 2011].

Smith III, J.O. (2007) 'Fixed-Point FFTs and NFFTs', in III, J.O.S. *Mathematics of the Discrete Fourier Transform (DFT)*, Available: http://www.dsprelated.com/dspbooks/mdft/Fixed_Point_FFTs_NFFTs.html [30 Oct 2011].

Ullmann, R.F. (2010) *An Algorithm for the Fast Hartley Transform*, 04 Jun, [Online], Available: http://sepwww.stanford.edu/theses/sep38/38_29_abs.html [01 Dec 2011].

Wikipedia (2005) *Window function*, 15 Dec, [Online], Available: http://en.wikipedia.org/wiki/Window_function [30 Mar 2012].

Wikipedia (2008) *Note - Wikipedia*, [Online], Available: http://en.wikipedia.org/wiki/File:Frequency_vs_name.svg [13 Oct 2011].

William Collins Sons & Co. Ltd. (2009) *Collins English Dictionary - Complete & Unabridged 10th Edition*, London: HarperCollins Publishers.

Appendices

1 Version Control

Git was chosen for version control during software development. This was used in conjunction with github for remote backup of the repository. All source code for the project is publicly available:

https://github.com/PeteHemery/Final-Year-Project

There is also a github generated front page for the project:

http://petehemery.github.com/Final-Year-Project/

This lists the Propeller Tool generated archive files and the repository available for download.

2 Variable FFT Spectrograph Test Results

Performance testing for the spectrograph involved modifying the constant assignments in the object files, recompiling and downloading to the board and reading timing information from the serial console.

The values of the constants modified in each object:

Spectrograph.spin	num_of_ffts	(1-4)
fft.spin	BITS_NN	(9-11)
sampler.spin	KHz	(4-45)

The Propeller Tool offers information about the size of the compiled program. These have been included for comparison.

Pass and Fail is determined on whether the sampler object stalls because there is no FFT available to accept the current sample buffer.

2.1 One FFT

Number of FFTs	FFT Sample Size	Sample Rate KHz	Speed of FFTs (ms)	Speed of Audio Cog Max (ms)	Speed of Audio Cog Min (ms)	Pass/ Fail	Program Size (Longs) Program	Program Size (Longs) Variable	Program Size (Longs) Free
1	512	4	36	163	127	Pass	1,450	3,072	3,666
1	512	5	36	138	102	Pass	1,450	3,072	3,666
1	512	6	36	121	85	Pass	1,450	3,072	3,666
1	512	7	36	109	72	Pass	1,450	3,072	3,666
1	512	8	36	100	100	Pass	1,450	3,072	3,666
1	512	9	36	92	92	Pass	1,450	3,072	3,666
1	512	10	36	87	87	Pass	1,450	3,072	3,666
1	512	11	36	82	46	Pass	1,450	3,072	3,666
1	512	12	36	78	78	Pass	1,450	3,072	3,666
1	512	13	36	75	39	Pass	1,450	3,072	3,666
1	512	14	36	72	72	Pass	1,450	3,072	3,666
1	512	15	36	70	34	Fail	1,450	3,072	3,666
1	1024	4	71	327	255	Pass	1,450	3,584	3,154
1	1024	5	71	276	204	Pass	1,450	3,584	3,154
1	1024	6	71	242	170	Pass	1,450	3,584	3,154
1	1024	7	71	217	146	Pass	1,450	3,584	3,154
1	1024	8	71	199	127	Pass	1,450	3,584	3,154
1	1024	9	71	185	113	Pass	1,450	3,584	3,154
1	1024	10	71	173	173	Pass	1,450	3,584	3,154
1	1024	11	71	146	92	Pass	1,450	3,584	3,154
1	1024	12	71	156	85	Pass	1,450	3,584	3,154
1	1024	13	71	150	150	Pass	1,450	3,584	3,154
1	1024	14	71	144	144	Pass	1,450	3,584	3,154
1	1024	15	71	139	68	Fail	1,450	3,584	3,154
1	2048	4	148	660	660	Pass	1,450	4,608	2,130
1	2048	5	148	557	557	Pass	1,450	4,608	2,130
1	2048	6	148	489	489	Pass	1,450	4,608	2,130
1	2048	7	148	440	440	Pass	1,450	4,608	2,130
1	2048	8	148	404	404	Pass	1,450	4,608	2,130
1	2048	9	148	375	375	Pass	1,450	4,608	2,130
1	2048	10	148	353	353	Pass	1,450	4,608	2,130
1	2048	11	148	334	186	Fail	1,450	4,608	2,130
1	2048	12	148	318	170	Fail	1,450	4,608	2,130
1	2048	13	148	305	157	Fail	1,450	4,608	2,130
1	2048	14	148	294	146	Fail	1,450	4,608	2,130
1	2048	15	148	284	284	Fail	1,450	4,608	2,130

2.2 Two FFTs

Number of FFTs	FFT Sample Size	Sample Rate KHz	Speed of FFTs (ms)	Speed of Audio Cog Max (ms)	Speed of Audio Cog Min (ms)	Pass/ Fail	Program Size (Longs) Program	Program Size (Longs) Variable	Program Size (Longs) Free
2	512	4	36	127	127	Pass	1,451	3,592	3,145
2	512	5	36	102	102	Pass	1,451	3,592	3,145
2	512	6	36	85	85	Pass	1,451	3,592	3,145
2	512	7	36	72	72	Pass	1,451	3,592	3,145
2	512	8	36	63	63	Pass	1,451	3,592	3,145
2	512	9	36	56	56	Pass	1,451	3,592	3,145
2	512	10	36	51	51	Pass	1,451	3,592	3,145
2	512	11	36	46	46	Pass	1,451	3,592	3,145
2	512	12	36	42	42	Pass	1,451	3,592	3,145
2	512	13	36	39	39	Pass	1,451	3,592	3,145
2	512	14	36	36	36	Pass	1,451	3,592	3,145
2	512	15	36	36	34	Fail	1,451	3,592	3,145
2	1024	4	71	255	255	Pass	1,451	4,616	2,121
2	1024	5	71	204	204	Pass	1,451	4,616	2,121
2	1024	6	71	170	170	Pass	1,451	4,616	2,121
2	1024	7	71	146	146	Pass	1,451	4,616	2,121
2	1024	8	71	127	127	Pass	1,451	4,616	2,121
2	1024	9	71	113	113	Pass	1,451	4,616	2,121
2	1024	10	71	102	102	Pass	1,451	4,616	2,121
2	1024	11	71	92	92	Pass	1,451	4,616	2,121
2	1024	12	71	85	85	Pass	1,451	4,616	2,121
2	1024	13	71	78	78	Pass	1,451	4,616	2,121
2	1024	14	71	73	73	Pass	1,451	4,616	2,121
2	1024	15	71	71	68	Fail	1,451	4,616	2,121
2	2048	4	148	511	511	Pass	1,452	6,664	72
2	2048	5	148	409	409	Pass	1,452	6,664	72
2	2048	6	148	341	341	Pass	1,452	6,664	72
2	2048	7	148	292	292	Pass	1,452	6,664	72
2	2048	8	148	255	255	Pass	1,452	6,664	72
2	2048	9	148	227	227	Pass	1,452	6,664	72
2	2048	10	148	204	204	Pass	1,452	6,664	72
2	2048	11	148	186	186	Pass	1,452	6,664	72
2	2048	12	148	170	170	Pass	1,452	6,664	72
2	2048	13	148	157	157	Pass	1,452	6,664	72
2	2048	14	148	148	146	Fail	1,452	6,664	72
2	2048	15	148	148	136	Fail	1,452	6,664	72

2.3 Three FFTs

Number of FFTs	FFT Sample Size	Sample Rate KHz	Speed of FFTs (ms)	Speed of Audio Cog Max (ms)	Speed of Audio Cog Min (ms)	Pass/ Fail	Program Size (Longs) Program	Program Size (Longs) Variable	Program Size (Longs) Free
3	512	16	36	31	31	Pass	1,452	4,112	2,624
3	512	18	36	28	28	Pass	1,452	4,112	2,624
3	512	20	36	25	25	Pass	1,452	4,112	2,624
3	512	22	36	23	23	Pass	1,452	4,112	2,624
3	512	24	36	21	21	Pass	1,452	4,112	2,624
3	512	25	36	20	20	Pass	1,452	4,112	2,624
3	512	26	36	19	19	Pass	1,452	4,112	2,624
3	512	27	36	18	18	Pass	1,452	4,112	2,624
3	512	28	36	18	18	Pass	1,452	4,112	2,624
3	512	29	36	18	17	Fail	1,452	4,112	2,624
3	512	30	36	19	17	Fail	1,452	4,112	2,624
3	1024	16	71	63	63	Pass	1,452	5,648	1,088
3	1024	18	71	56	56	Pass	1,452	5,648	1,088
3	1024	20	71	51	51	Pass	1,452	5,648	1,088
3	1024	22	71	46	46	Pass	1,452	5,648	1,088
3	1024	24	71	42	42	Pass	1,452	5,648	1,088
3	1024	25	71	40	40	Pass	1,452	5,648	1,088
3	1024	26	71	39	39	Pass	1,452	5,648	1,088
3	1024	27	71	37	37	Pass	1,452	5,648	1,088
3	1024	28	71	36	36	Pass	1,452	5,648	1,088
3	1024	29	71	36	35	Fail	1,452	5,648	1,088
3	1024	30	71	37	34	Fail	1,452	5,648	1,088

2.4 Four FFTs

Number of FFTs	FFT Sample Size	Sample Rate KHz	Speed of FFTs (ms)	Speed of Audio Cog Max (ms)	Speed of Audio Cog Min (ms)	Pass/ Fail	Program Size (Longs) Program	Program Size (Longs) Variable	Program Size (Longs) Free
4	512	15	36	34	34	Pass	1,452	4,632	2,104
4	512	20	36	25	25	Pass	1,452	4,632	2,104
4	512	25	36	20	20	Pass	1,452	4,632	2,104
4	512	30	36	17	17	Pass	1,452	4,632	2,104
4	512	35	36	14	14	Pass	1,452	4,632	2,104
4	512	40	36	12	12	Pass	1,452	4,632	2,104
4	512	41	36	12	12	Pass	1,452	4,632	2,104
4	512	42	36	12	12	Pass	1,452	4,632	2,104
4	512	43	36	12	11	Fail	1,452	4,632	2,104
4	512	44	36	13	11	Fail	1,452	4,632	2,104
4	512	45	36	13	11	Fail	1,452	4,632	2,104
4	1024	15	71	68	68	Pass	1,452	6,680	56
4	1024	20	71	51	51	Pass	1,452	6,680	56
4	1024	25	71	40	40	Pass	1,452	6,680	56
4	1024	30	71	34	34	Pass	1,452	6,680	56
4	1024	35	71	29	29	Pass	1,452	6,680	56
4	1024	40	71	25	25	Pass	1,452	6,680	56
4	1024	41	71	24	24	Pass	1,452	6,680	56
4	1024	42	71	24	24	Pass	1,452	6,680	56
4	1024	43	71	24	23	Fail	1,452	6,680	56
4	1024	44	71	25	23	Fail	1,452	6,680	56
4	1024	45	71	26	22	Fail	1,452	6,680	56

3 Example of Propeller Tool Documentation View Mode

```
***************************************
*  Microphone-to-VGA v1.0             *
*  Author: Chip Gracey                *
*  Copyright (c) 2006 Parallax, Inc.  *
*  See end of file for terms of use.  *
***************************************
Modified by Pete Hemery for Undergraduate Project 2011-2012
Displays detected frequency, note, octave and cent offset from microphone input.

Object "Freq_Counter_Demo" Interface:

PUB  start

Program:   2,091 Longs
Variable:    349 Longs

__________
PUB  start

           n / 12
  f = (2 ^        ) x 440 Hz

  note = (log2(f / 440)) x 12

  This uses A4 as the centre octave of 0. So offset by 4.

  octaves automatically yield factors of two times the original frequency,
   since n is therefore a multiple of 12
   (12k, where k is the number of octaves up or down), and so the formula reduces
to:

           12k / 12                  k
  f = (2 ^          ) x 440   =   (2 ^   ) x 440

  code:
  lnote = log2(f / 440)

  octave = lnote + 4

  note = ( octave - truncated (octave) ) * 12

  cents = (note - truncated (note) ) * 100
```

(Licence size intentionally reduced)

4 Additional Spin and Propeller Assembler Syntax Definitions

4.1 Symbol Rules

Symbols are case-insensitive, alphanumeric names either created by the compiler (reserved word) or by the code developer (user-defined word). They represent values (constants or variables) to make source code easier to understand and maintain. Symbols must fit the following rules:

1) Begins with a letter (a – z) or an underscore '_'.
2) Contains only letters, numbers, and underscores (a – z, 0 – 9, _); no spaces allowed.
3) Must be 30 characters or less.
4) Is unique to the object; not a reserved word or previous user-defined symbol.

4.2 Syntax Symbols

`%` Binary number indicator, as in `%1010`.

`%%` Quaternary number indicator, as in `%%2130`.

`$` Hexadecimal indicator, as in `$1AF` or assembly 'here' indicator.

`"` String designator `"Hello"`.

`_` Group delimiter in constant values, or underscore in symbols.

`#` Object-Constant reference: `obj#constant`.

`.` Object-Method reference: `obj.method(param)` or decimal point.

`..` Range indicator, as in `0..7`.

`:` Return separator: `PUB method : sym`, or object assignment, etc.

`|` Local variable separator: `PUB method | temp, str`.

`\` Abort trap, as in `\method(parameters)`.

`,` List delimiter, as in `method(param1, param2, param3)`.

`( )` Parameter list designators, as in `method(parameters)`.

`[ ]` Array index designators, as in `INA[2]`.

`{ }` In-line/multi-line code comment designators.

`{{ }}` In-line/multi-line document comment designators.

`'` Code comment designator.

`' '` Document comment designator.

4.3 Flow Control

One of the important features of Spin is the use of whitespace. The Propeller Tool displays a coloured arrow to show which statements below to which preceding conditional statement.

`IF` Conditionally execute one or more blocks of code.

```
...ELSEIF
...ELSEIFNOT
...ELSE
```

`IFNOT` Conditionally execute one or more blocks of code.

```
...ELSEIF
...ELSEIFNOT
...ELSE
```

`CASE` Evaluate expression and execute block of code that satisfies a condition.

```
...OTHER
```

`REPEAT` Execute block of code repetitively an infinite or finite number of times with optional loop counter, intervals, exit and continue conditions.

```
...FROM
...TO
...STEP
...UNTIL
...WHILE
```

`NEXT` Skip rest of `REPEAT` block and jump to next loop iteration.

`QUIT` Exit from `REPEAT` loop.

`RETURN` Exit `PUB/PRI` with normal status and optional return value.

`ABORT` Exit `PUB/PRI` with abort status and optional return value.

4.4 Unary Operators

`+` Positive (+X); unary form of Add.

`-` Negate (-X); unary form of Subtract.

`--` Pre-decrement (--X) or post-decrement (X--) and assign.

`++` Pre-increment (++X) or post-increment (X++) and assign.

`^^` Square root.

`||` Absolute Value.

`~` Sign-extend from bit 7 (~X) or post-clear to 0 (X~).

`~~` Sign-extend from bit 15 (~~X) or post-set to -1(X~~).

`?` Random number forward (?X) or reverse (X?).

`|<` Decode value (modulus of 32; 0-31) into single-high-bit long.

`>|` Encode long into magnitude (0 - 32) as high-bit priority.

`!` Bitwise: NOT.

`NOT` Boolean: NOT (promotes non-0 to -1).

`@` Symbol address.

`@@` Object address plus symbol value.

4.5 Binary Operators

NOTE: All right-column operators are assignment operators.

`=` --and-- `=` Constant assignment (`CON` blocks).
`:=` --and-- `:=` Variable assignment (`PUB/PRI` blocks).
`+` --or-- `+=` Add.
`-` --or-- `-=` Subtract.
`*` --or-- `*=` Multiply and return lower 32 bits (signed).
`**` --or-- `**=` Multiply and return upper 32 bits (signed).
`/` --or-- `/=` Divide (signed).
`//` --or-- `//=` Modulus (signed).
`#>` --or-- `#>=` Limit minimum (signed).
`<#` --or-- `<#=` Limit maximum (signed).
`~>` --or-- `~>=` Bitwise: Shift arithmetic right.
`<<` --or-- `<<=` Bitwise: Shift left.
`>>` --or-- `>>=` Bitwise: Shift right.
`<-` --or-- `<-=` Bitwise: Rotate left.
`->` --or-- `->=` Bitwise: Rotate right.
`><` --or-- `><=` Bitwise: Reverse.
`&` --or-- `&=` Bitwise: AND.
`|` --or-- `|=` Bitwise: OR.
`^` --or-- `^=` Bitwise: XOR.
`AND` --or-- `AND=` Boolean: AND (promotes non-0 to -1).
`OR` --or-- `OR=` Boolean: OR (promotes non-0 to -1).
`= =` --or-- `= = =` Boolean: Is equal.
`<>` --or-- `<>=` Boolean: Is not equal.
`<` --or-- `<=` Boolean: Is less than (signed).
`>` --or-- `>=` Boolean: Is greater than (signed).
`=<` --or-- `=<=` Boolean: Is equal or less (signed).
`=>` --or-- `=>=` Boolean: Is equal or greater (signed).

The variable assignment operator '`:=`' can be used for multiple assignments simultaneously. It can also be used inside expressions for intermediate results. For example:

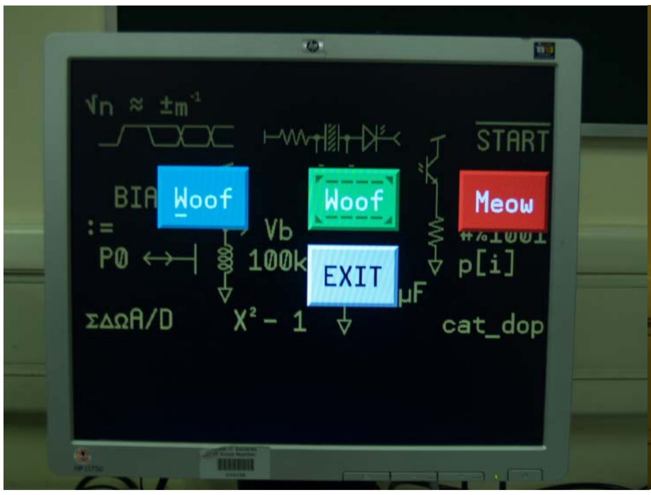
```
    TripleCopy := Triple := 1.5 * (Double := (Temp := 1) * 2 )
```

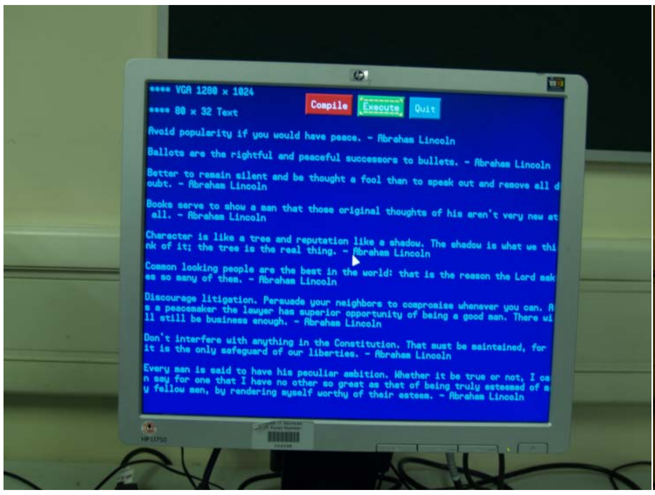
Many of the binary operators have shorthand equivalents that allow multiple actions to be performed based on the values of the given variables.

5 Propeller Graphical Demos

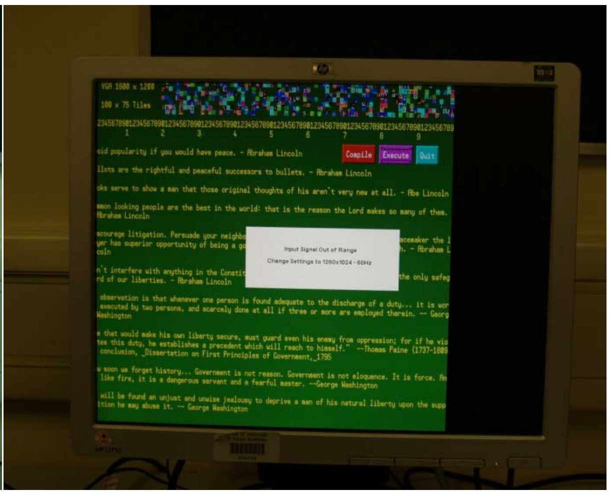
5.1 Parallax VGA Demos

.





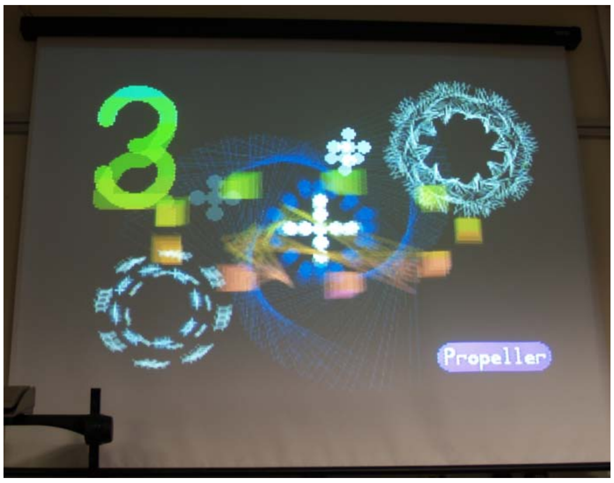
```
**** VGA 1280 x 1024
**** 80 x 32 Text
Avoid popularity if you would have peace. - Abraham Lincoln
Ballots are the rightful and peaceful successors to bullets. - Abraham Lincoln
Better to remain silent and be thought a fool than to speak out and remove all d
oubt. - Abraham Lincoln
Books serve to show a man that those original thoughts of his aren't very new at
 all. - Abraham Lincoln
Character is like a tree and reputation like a shadow. The shadow is what we thi
nk of it; the tree is the real thing. - Abraham Lincoln
Common looking people are the best in the world: that is the reason the Lord mak
es so many of them. - Abraham Lincoln
Discourage litigation. Persuade your neighbors to compromise whenever you can. A
s a peacemaker the lawyer has superior opportunity of being a good man. There wi
ll still be business enough. - Abraham Lincoln
Don't interfere with anything in the Constitution. That must be maintained, for
it is the only safeguard of our liberties. - Abraham Lincoln
Every man is said to have his peculiar ambition. Whether it be true or not, I ca
n say for one that I have no other so great as that of being truly esteemed of m
y fellow men, by rendering myself worthy of their esteem. - Abraham Lincoln
```

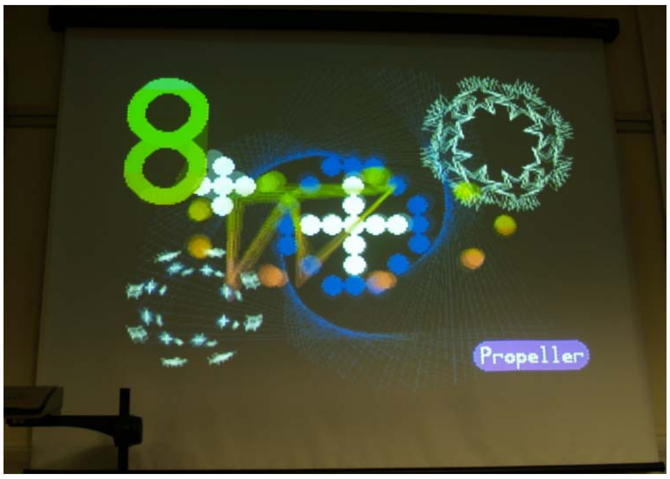


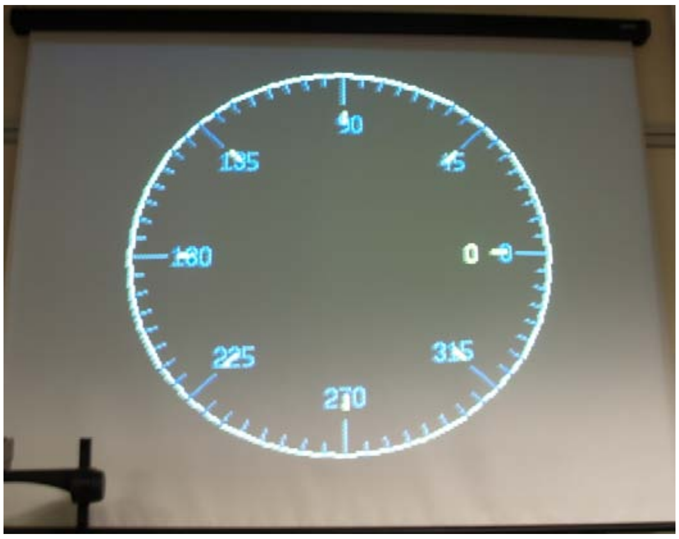
```
VGA 1600 x 1200
100 x 75 Tiles
234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
         1         2         3         4         5         6         7         8         9
oid popularity if you would have peace. - Abraham Lincoln
llots are the rightful and peaceful successors to bullets. - Abraham Lincoln
oks serve to show a man that those original thoughts of his aren't very new at all. - Abe Lincoln
mmon looking people are the best in the world: that is the reason the Lord makes so many of them.
Abraham Lincoln
scourage litigation. Persuade your neighbo                                acemaker the l
yer has superior opportunity of being a go                               h. - Abraham L
coln
n't interfere with anything in the Constit                               the only safeg
rd of our liberties. - Abraham Lincoln
 observation is that whenever one person is found adequate to the discharge of a duty... it is wor
 executed by two persons, and scarcely done at all if three or more are employed therein. -- Georg
Washington
e that would make his own liberty secure, must guard even his enemy from oppression; for if he vio
tes this duty, he establishes a precedent which will reach to himself."  --Thomas Paine (1737-1809
 conclusion, _Dissertation on First Principles of Government,_1795
w soon we forget history... Government is not reason. Government is not eloquence. It is force. An
 like fire, it is a dangerous servant and a fearful master. --George Washington
 will be found an unjust and unwise jealousy to deprive a man of his natural liberty upon the supp
ition he may abuse it. -- George Washington
```

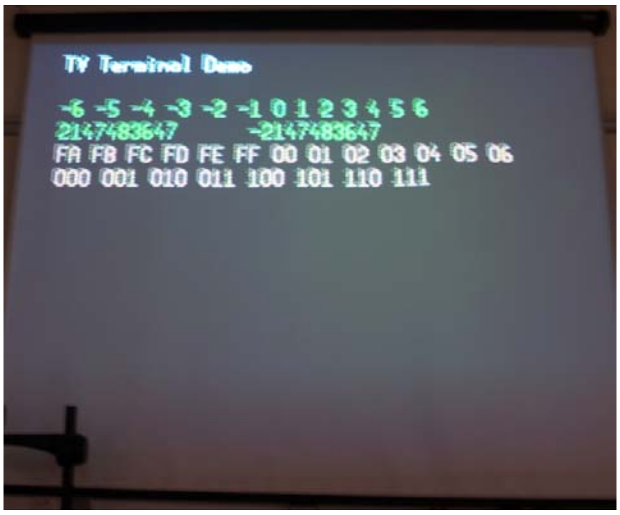
```
Input Signal Out of Range
Change Settings to 1280x1024 - 60Hz
```

5.2 Parallax TV Demos

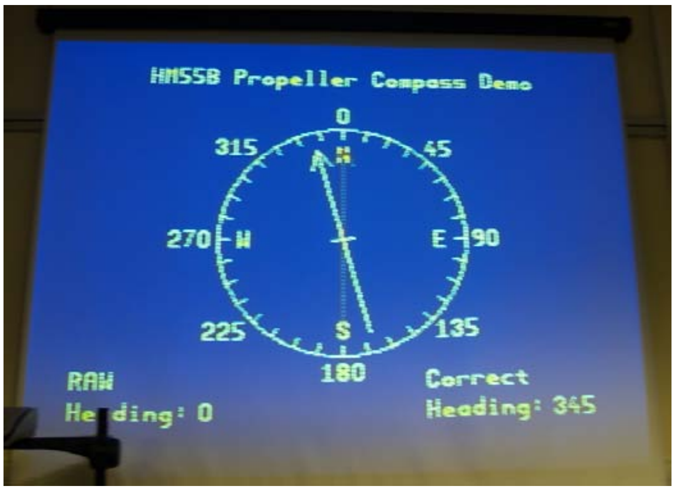




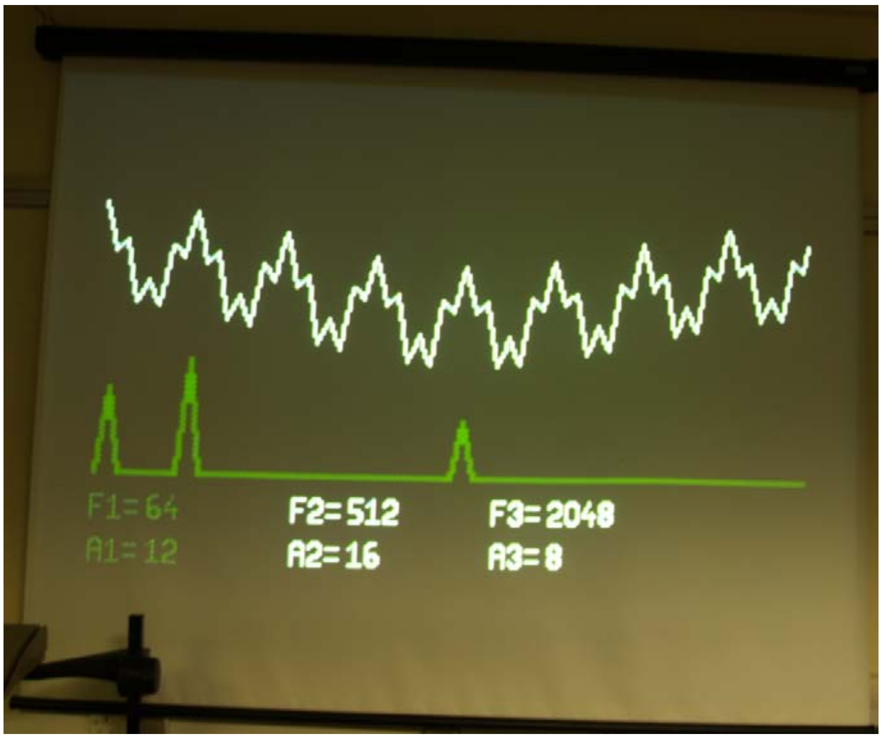


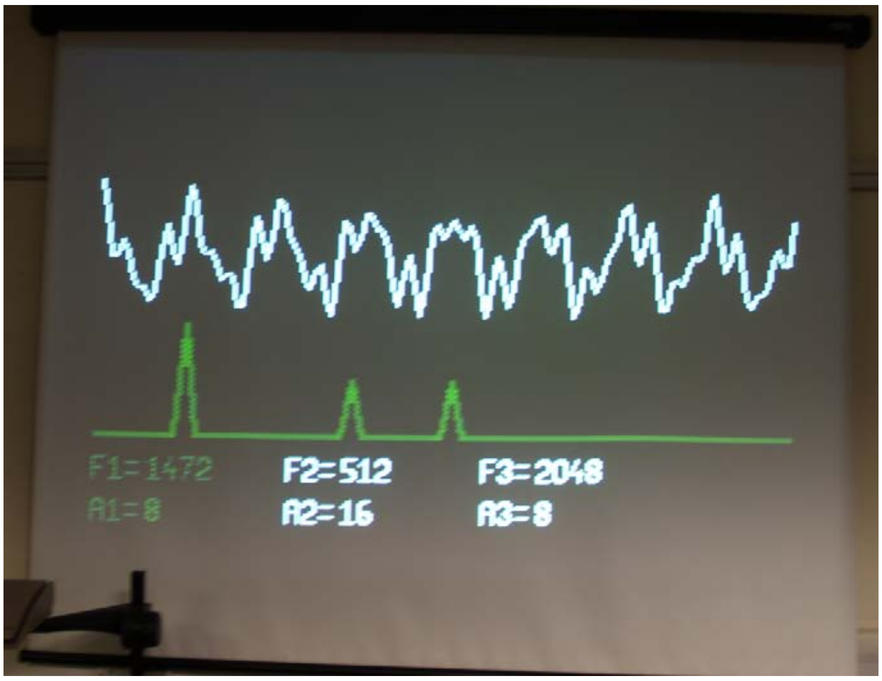


```
TV Terminal Demo
-6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6
2147483647        -2147483647
FA FB FC FD FE FF 00 01 02 03 04 05 06
000 001 010 011 100 101 110 111
```



5.3 Fast Hartley Transform Demo





6 Source Code CD