How To Develop Embedded Software

A Case Study

Exopiped Press

Table of Contents

INTRODUCTION

This book is a case study of an embedded software project. It describes an actual project from start to finish, and all of the material and documents developed and used along the way.

This information supplied comes directly from the author's 30 years of experience developing embedded software for electronic products made by top U.S. corporations.

The example system is a touch-pad synthesizer, similar to a Theremin. The original analog Theremin responded to user hand motions near its two antennae, by producing musical tones of varying pitch and volume.

The digital version accepts a user's touch on a touchscreen or XY pad, and responds with notes of varying pitch and volume. This simple example is challenging enough to illustrate some common embedded development practices.

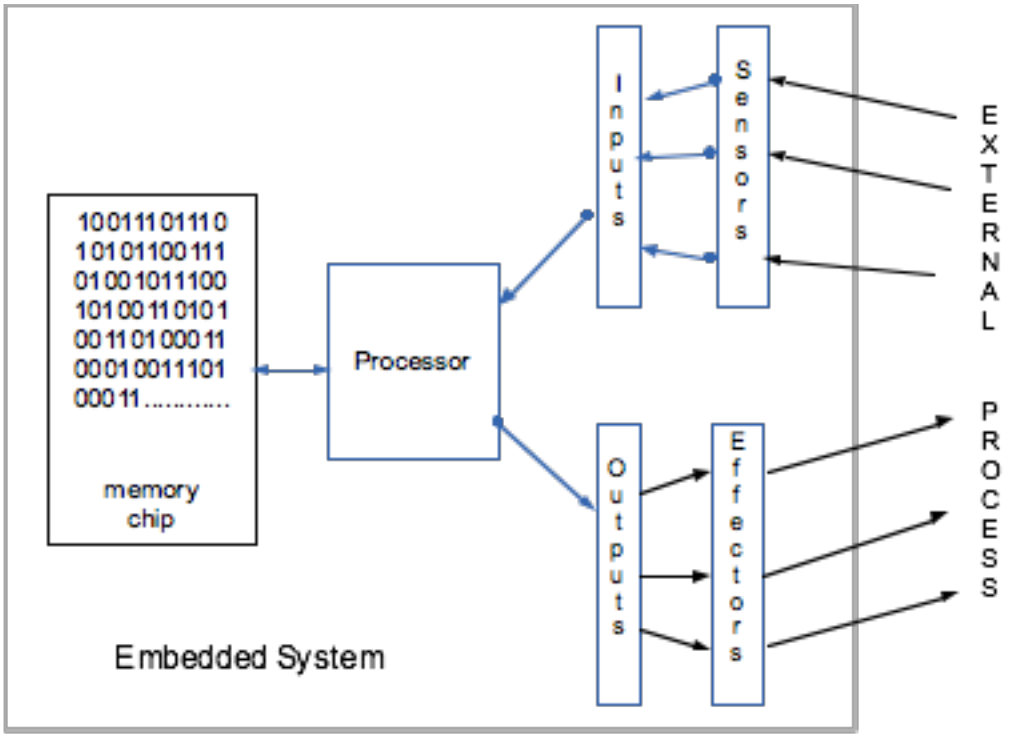
This book is written with three types of readers in mind. First is the experienced PC programmer who wishes to transition into the world of embedded systems.

Another is the electronic engineer who writes code but has not yet had much exposure the sofware side of architecture and design.

Finally, the hobbyist, who works well within the Arduino or similar environment, can learn additional concepts that will help him take the next step into software engineering.

SOME DEFINITIONS

An **embedded system** is an information appliance that is supplied with sensors and effecters which permit it to interact with the external environment. It uses a microprocessor to control the sensors and effecters, and software to guide its interpretation of incoming signals, and its production of outgoing signals.



The memory chips, processor, input and output peripherals, sensors and effecters comprise the hardware of the embedded system. The bit sequence in the memory chip is called the software of the embedded system.

Embedded systems are collections of interacting components. The definitions below provide a framework for discussing those collections further.

Every component of an embedded system which interacts with other

components is called an **object**. Each object is distinct from other objects, has an internal state, and exhibits a repertoire of behavior. Objects exhibit their behavior by exchanging messages with other objects. A **message** is an influence that passes between objects.

Objects may be hardware or virtual. A **hardware object** is made out of material parts. The messages it exchanges are typically electrical or mechanical signals. A **virtual object** is simulated in software, and exchanges messages by invoking methods or functions in other virtual objects, or setting registers in hardware objects.

A **compound object** is an object which is composed of other objects. A **mixed object** is a compound object composed of some hardware and some virtual objects.

A **class** is a collection of one or more virtual objects, which respond to the same messages (have the same repertoire of behavior), but which may have different internal states. Virtual objects respond to messages by executing **methods**, or functions, which are part of the code base of the class.

You may want to consult other sources for further explanation of some of the concepts.

See, for example, Chapter 3, Classes and Objects, in Grady Booch's book Object Oriented Design[2].

THE CONCEPTUAL MAP

Software developers create bit sequences that go into memory chips.

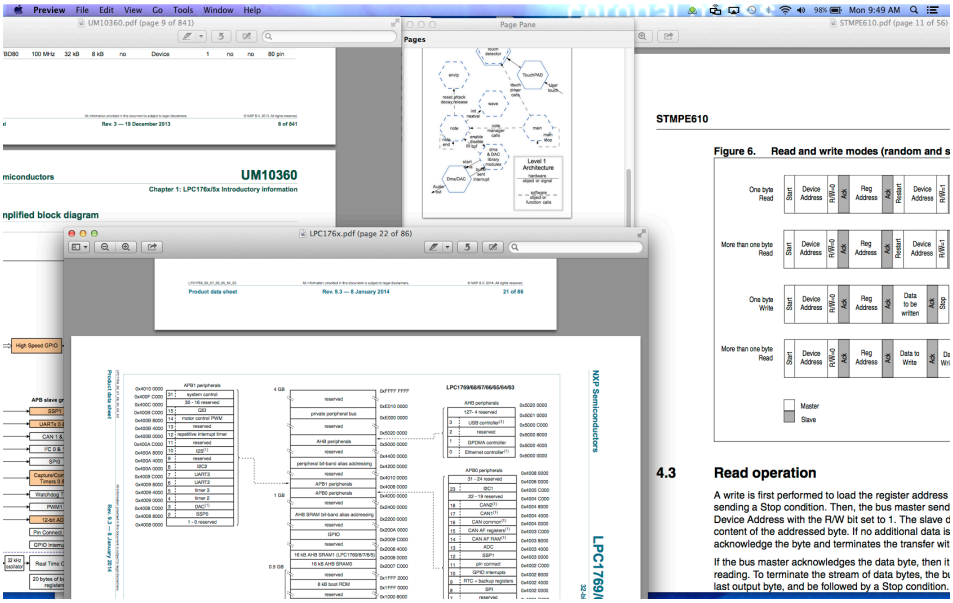
The typical developer cannot listen to a description of desired system behavior, and then just dash off a bit sequence that does the trick. Instead, he must create a conceptual map of the problem to be solved, and all of the elements that make up a solution. The conceptual map has both a documentary and a mental aspect.

The Documentary Aspect of the Conceptual Map

The documentary aspect of the conceptual map is a collection of data sheets, websites, analysis and design documents, interviews, and meeting minutes which bear on the project. These documents are sometimes organized in a project folder that makes them readily accessible to each member of the team.

In the earlier years the documentary map looked like this:

In recent times, the documentary map looks more like this:



The Mental Aspect of the Conceptual Map

The mental aspect of the conceptual map is a representation in each team member's mind of the personnel, environment, goals, hardware, and software objects involved in a development project, and the ways in which they interact.

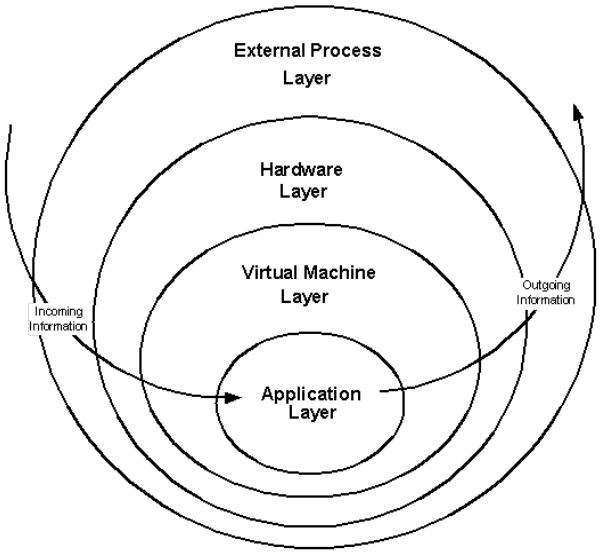
Pessimistic Impression of the mental conceptual map

The dreaded "learning curve" of an embedded project is the time necessary for a new team member to create, from conversations with team members and from the project's documentary map, a mental map for the project.

Layers of the Conceptual Map

It can be helpful to subdivide the conceptual map into different layers of concepts, each of which deals with different portions of the mapping between the external world, and the embedded software which interacts with it.

The diagram below shows one such subdivision of a project's conceptual map:



In this diagram, the subdivisions of the conceptual map relate to the flow of information or influence in the embedded system.

Influence flows in from the external process layer through the hardware layer, where it is transformed into information, which flows into the virtual machine layer, and is processed by the application layer.

Response information is sent back out via the virtual machine layer, and is transformed by the hardware layer into influence on the external process layer.

In the **external process layer**, the embedded system is considered as a unit which exchanges influences with the external world. The concepts in this

layer describe in a general way each of those interactions. They also describe in measurable detail every aspect of those interactions which motivated the building of the system.

The **hardware layer** of the conceptual map contains data sheets, schematics, and concepts related to the operation of the embedded system hardware. Documentation of the hardware is provided by hardware vendors and designers for the processor, peripherals, sensors, effecters, memory chips, comm chips, and power supplies.

The layer between the hardware and the application is called the **virtual machine layer**. It describes the virtual (software-simulated) objects through which the application accesses the hardware, and organizes itself in time. These virtual objects include the primitives of the programming language, the operating system (if any), and any code libraries supplied with the compiler or operating system or written separately by team members.

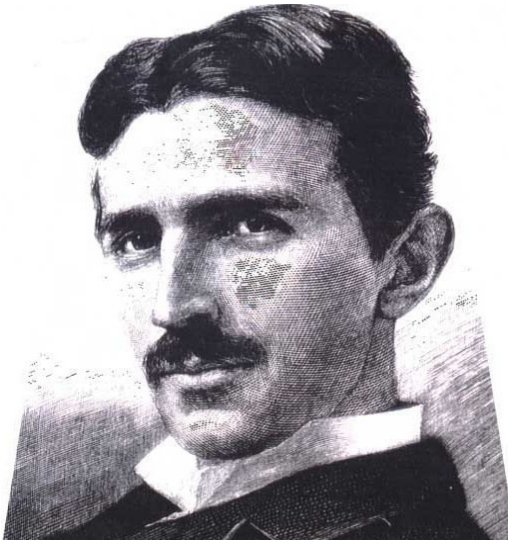
The **application layer** is a collection of virtual objects which interact to model the external process and/or exchange influences with it via the virtual machine and hardware layers. This layer is mostly invented and documented by the developers. It may also include virtual objects supplied in code libraries by the system sponsor, or by third party vendors.

The quality of the conceptual map, in both its physical and mental forms, figures prominently in the success or failure of an embedded project.

If the team members have created a well organized collection of documents, and if they take time to read and understand them; the project has a better prospect of success.

REAL ENGINEERS

A belief persists among some developers, typically those more hardware oriented, that real engineers don't write documentation. There is some justification for this position. For a small system, one not too complex; it is possible to write code without extensively documenting the conceptual map.



Nikola Tesla -- Photo Courtesy of Tesla Memorial Society of New York
www.teslasociety.com

Nikola Tesla, arguably the Earth's greatest engineer, is widely believed to have kept most of his plans and designs in his head. He even managed to set simulations running in his mind, and revisit them from time to time to check for wear and tear on components.

Mr. Tesla was, without a doubt, a master of the mental conceptual map. Even so, he kept extensive lab notes, and did in fact document his work in patent applications.

If the choice is made not to document, troubles arise when any of the following conditions are met. The number of callable functions exceeds a

hundred, there is frequent turnover in the engineering staff, or safety and regulatory issues require full disclosure of the development process.

Throughout this book, various methods will be used to create and document the conceptual map, be it in requirements, architecture, or design documents. If the reader is convinced that documenting gilds the lily, then he will probably ignore these methods.

Just as some engineers favor sparse documentation, others prefer rich documentation. The latter group are persons who require more rigor than that supplied in the methods used here. Those readers will no doubt feel free to use their own methods of documenting the conceptual map, or adopt methods that may be found in abundance in the literature.

DEVELOPMENT PROCESSES

The software developer's task is to complete the conceptual map of the embedded system, and to implement and test it's virtual objects on the target hardware. This leads him to discover and document the system's required interactions with the outer world, learn how the hardware works, understand the virtual machine, invent the interacting virtual objects of the application, debug, test, and validate the system. To these ends, the developer participates in the processes described below.

AQUAINTANCE

Become familiar with the physical and social environment in which the system will be developed.

REQUIREMENTS ANALYSIS

Develop descriptions of the external process and the desired useful ways of interacting with it. Formulate a complete set of measurable goals for the embedded system to meet. Digest the vendor and hardware designer supplied documentation of the hardware and virtual machine. Understand any predecessor systems. If system safety is a concern, perform an initial study of the hazards created by the system, and possible ways of mitigating those hazards.

ARCHITECTURE

Describe in overview how the hardware and virtual objects work together to interact with the external process. Put the description into an architectural document. Revise analysis concepts where necessary to accommodate real-world facts revealed by the architecture.

ESTIMATING

For planning purposes, break the remaining software development process into small chunks, and estimate how much effort of how many persons will be required to complete each chunk. Make some guesses about the number of lines of code, and function points required, and estimate total

project resources from those estimates.

Tool Selection

Choose and install tools to be used in the project. Choice of tools depends on the processor(s) chosen, as well as the number of different computers involved in the system. Most systems will require at least a text editor and compiler or interpreted language, and some kind of debugger.

Hardware Support

Support the persons developing the embedded system hardware by writing test code. Use the information gained from writing the test code to improve the architecture and assist with the design of the system. In some cases incorporate the test code directly into the final system code.

Design

Create an orderly description of a collection of virtual objects, which implement the system architecture. Document the collection in a design document. Review the design document with team members and third parties. Make sure the design will result in a system that meets its goals and operates safely. Revise analysis and architectural concepts as necessary to accommodate changes motivated by the design process.

Code

Implement in the chosen languages, all invented virtual objects, including those in the virtual machine. Revise analysis, architecture, and design as needed.

Debug

Make all of the hardware and virtual objects work as intended. Revise analysis, architecture, design and code as needed.

Integrate

Make all of the hardware and virtual objects interact in the way envisioned in the architecture and design. Revise analysis, architecture, design and code as needed.

VERIFY

Demonstrate that the virtual objects work properly. Repeat above activities as needed.

VALIDATE

Formally demonstrate that the overall embedded system meets the goals it was intended to meet. Repeat above activities as needed.

The processes described above will be considered in more detail in the coming sections.

ACQUAINTANCE

Whether you come in at the beginning or some time during the middle of a project, your first task is to learn everything there is to know about the project. In no particular order, you need to know:

1) Why is this project happening?
2) What persons support the project?
3) What persons oppose the project?
4) Is this a new product, or a new version of an older product?
5) What is the technical environment of the project?
6) Are project resources adequate to support your efforts?
7) What is the expected schedule for the project? Is it realistic?
8) What are the skills, strengths, and weaknesses of other team members?
9) Which team members do you like? Which ones can you trust? What are their skill sets?
10) Is product safety an issue? If so, does the resource provider support an emphasis on product safety?

You won't answer all these questions immediately. In fact, you may never answer all of these questions; but these are things you should find out as soon as possible after coming into a new project.

Now you may say: "Look man, I'm just a programmer on this project. I do my work, they write my check. What do I care about all that political stuff?"

That may be so, but you are also responsible for your own life, and a medium sized embedded project is going to account for a big chunk of the next year or two of your life. It's going to affect your relationships, and your happiness for as long as you are involved. Better to know what you are walking into, than to stumble blindly into a disaster in the making.

Why is this project happening? Ask around. Normally, you will find someone who champions the project. Get to know that person. What is their motivation? If they get a spark in their eye when they talk about the

project, that is a good sign. If they are just occupying an organizational position and carrying out policy, that may still be OK. If they are looking to make a lot of money, that is probably a bad sign. There are much easier ways to make money.

It's can be tough at first to discover, but you need to know the primate power relationships within the organization, as they relate to the project champion. He may be on the way out, in which case you'll have to find another situation. Make sure the project champion has the support of the main monkeys within the organization.

Next feel out the network of technical personnel. Have they been with the organization a long time, or did they just arrive. If the latter, are they replacing people who just left? Why? How is the morale of the technical staff? If they spend more time talking about the organization than about the work, get out of there fast.

Does your entry into the project ignite professional jealousy in anyone on the staff? If so, acquaint yourself with that person. Once you get to know each other, the problem will likely go away. You will have a new friend. If that doesn't happen, keep your eye on that person.

When you are comfortable with the project motivation and personnel, you are ready to enjoy the first technical challenge: discovering the requirements.

REQUIREMENTS ANALYSIS

Before you can design your software, you need to find out what it is supposed to do. You accomplish this by filling in the external process layer of the conceptual map. This activity is usually called requirements analysis.

If you are replacing an existing system, someone may tell you: "Make it work like the Blivitt system, only make it work better". Life is good. You just **reverse engineer** the Blivitt system, and you are halfway home.

On the other hand, there may be disagreement about what the system should do. You may have to hold a **requirements workshop** with major system stakeholders, and extract expectations in all of their diversity and conflict.

A user or client representative may come forward or be supplied and offer to work with you. Accept the offer. Be aware that other parties to the development may have ideas different from those of the supplied representative.

No matter how you gather the requirements, they can be documented for two purposes: to communicate to all interested parties what the system will do, and to provide measurable descriptions of all system features for architectural, design, and testing purposes.

Inasmuch as these are very different purposes, it is a good idea to use two different kinds of documentation. These are the **use case summary,** and the f**ormal requirements list**. In no case should you attempt to communicate requirements to non-engineers using the formal requirements list.

Sub-sections below give examples of a use case summary and a formal requirements list. Then the requirements gathering techniques of reverse engineering and the requirements workshop are described.

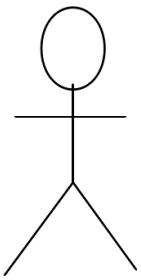
For a wealth of useful information on requirements analysis, including

details on the tools described here and much more, see Managing Software Requirements: A Unified Approach[3].

Use Case Summary

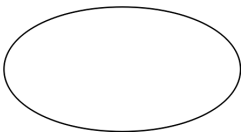
In his 1992 bestseller Object-Oriented Software Engineering[2], Ivar Jacobson documented a good way to describe the interactions of a software system with its external environment. He called it "use case analysis" (use is pronounced the same way as the first word in "Yous guys grab da dame.") It is particularly helpful for making a first stab at system requirements, and for communicating requirements to non-engineers.

First, one identifies the different agents which interact with the system. These agents Ivar called **actors**. A actor could be a person, another computer, an animal, a machine, or anything else interacting with the embedded system. Actors are represented graphically by the symbol:



Actor

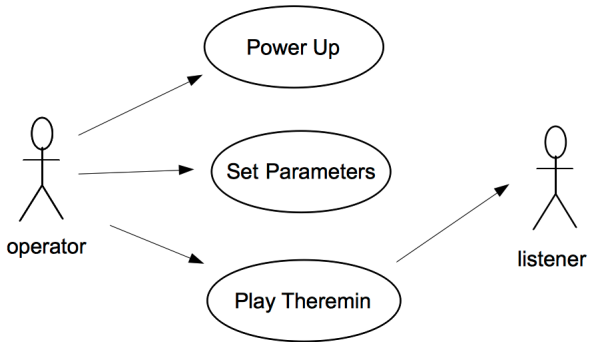
Next, one identifies a collection of situations, in which actors interact with the system. These situations, Ivar called use cases. Use cases are represented graphically by the symbol:



Use Case

In Ivar's approach, one simply lists all of the significant use cases for the embedded system, and the actors involved with each. A graphical overview of all of the use cases is optional.

The next graphic shows the use cases for the example embedded system, a digital Theremin:



Actors: Operator, Listener
Operator -- The user of the digital Theremin device
Listener -- Person listening to the output. May be the same person as the operator.

Use Cases: Power Up, Set Parameters, Play Theremin

Power Up -- The Operator plugs the digital Theremin into a USB socket to. This causes the embedded system to go through its initialization phase.

Set Parameters -- At any time after initialization, the user my alter the "hardness" or "softness" of the attack by using up or down movements of the joy-switch on the host circuit board. Moving the switch down shortens the attack phase of each note, making the attack harder. Moving the switch up lengthens the attack phase of each note, making the attack softer. In addition to controlling the attack phase of each note, the joy-switch may be used to control the release phase. By moving the joy-switch to the left the release phase (time during which the note dies away) is shortened. By moving the joy-switch to the right, the release phase is lengthened. Pushing the joy-switch in the center toggles the waveform used by each

note among the available alternative waveforms.

Play Theremin -- The Operator taps or strokes the touchpad to make musical notes varying in loudness and frequency. The loudness of the note can be increased by moving the touch in the direction of increasing X value. The frequency of the note can be increased by moving the touch in the direction of increasing Y value. Lifting the finger causes the note to release, eventually fading out. Each touch of the screen begins a new note.

That completes the use cases for our example. In more complex systems, there will be many more use cases. Some use cases may employ others as subroutines. See Dr. Jacobson's book for ways of diagramming many interacting use cases.

As in our example, it is often sufficient to forego the graphical representation of hte use cases, and to just write them as short descriptions of permissable user interactions.

Once you know what the system does, you can make a first draft of use cases for a medium sized embedded system (200-500 callable functions) in an afternoon. It may take several calendar weeks to pass the description around, hold meetings, and finally get agreement on the requirements documented by the use cases.

Formal Requirements List

A formal requirements list is a minimal, but exhaustive collection of succinct, testable statements about the operation of the embedded system and/or its software. It is often helpful to include photos, diagrams or tables in the formal requirements list.

Sometimes the formal software requirements are placed into their own document. Sometimes they are included in a separate section of the formal system requirements. Sometimes they are tagged as software requirements within the formal requirements document. Sometimes they

are kept in a requirements database or spreadsheet.

Why do you need a formal requirements list? It keeps all of the team members working toward the same goals. You will refer to it during the architecture and design processes. It will guide you in making tradeoffs. You will use it to write test plans that allow you to measure how well your work has turned out. The process of writing it will expose you to relevant issues you might not otherwise consider.

There are many recommended formats for requirements lists. There's an IEEE standard. There are U.S. government standards. There may be standards within the organization which is sponsoring your project. Don't fight over the format.

The only thing you need is a collection of sentences, tables, and/or diagrams, each of which makes a succinct, testable statement about the operation of the software. That can be put into any format necessary within context of your project.

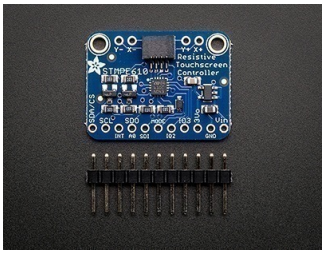
Below is a list of formal requirements for the digital Theremin example:

1) The hardware for the Theremin project has already been chosen, primarily for its ready availablity, and superior compiler and library support. It includes an mbed.org Microcontroller board with an LPC1768 ARM Cortex-M3 Processor.

That 40pin DIP board will be plugged into an mbed Application Board,

which provides additional peripherals, such as a five-position joy-switch, and pull-up resistors for a I2C lines SDA and SCL, which will connect to a touchscreen controller chip.

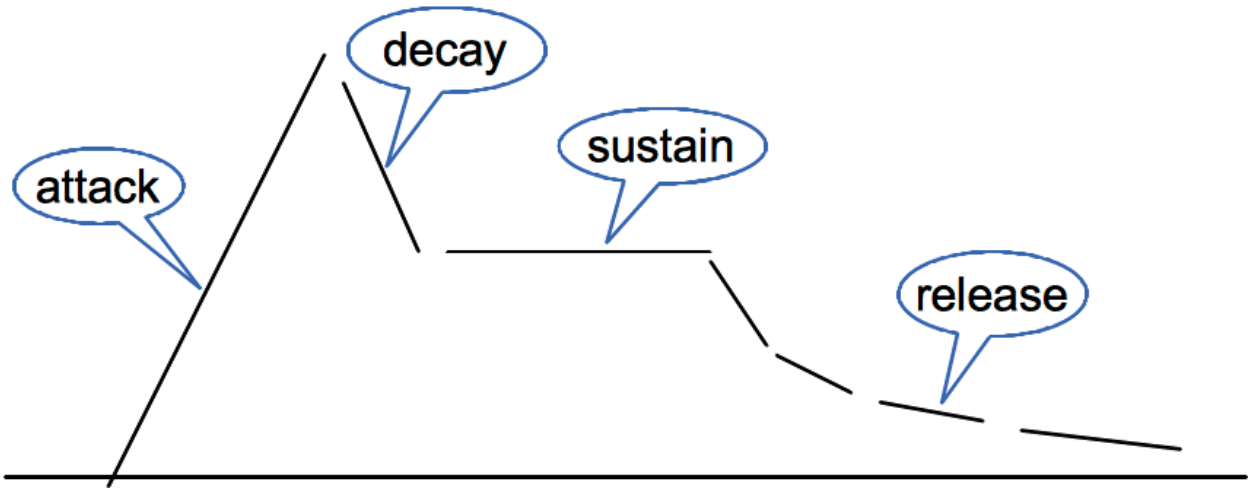
Both the processor board and application board may be purchased from www.sparkfun.com or www.adafruit.com. The touchscreen controller board uses the SMTPE610 controller chip, and is available from adafruit.com.



Any 4-wire resistive touchscreen can be used, but the one purchased for this example is roughly 2.2 (Y dimensiob) by 2.75 (X dimension) inches, and has a connector compatible with the touchscreen controller board. It is similar to the touchscreen, from adafruit.com, shown below:

2) When powered up, the Therimen briefly enters an initialization period, during which it sets up the system hardware and computes waveform table(s). Then it enters normal operation using a saw waveform as it's default.

3) Every note played consists of a waveform modulated by an envelope. The waveform has the frequency and volume determined by the Y-X touch position. The envelope modulation evolves in time through four separate stages: attack, decay, sustain, and release. The release stage begins when the touch is removed.



4) The Theremin accepts input from the resistive touchscreen and the joy-switch. It reacts to a touch by triggering a note of a frequency and volume determined by the Y & X position of the touch. It reacts to a center press of the joy-switch by making the next waveform available for the next note or notes. It reacts to left, right, up, and down movements of the joy-switch to shorten release phase, lengthen release phase, lengthen attack phase, and shorten attack phase of the next note or notes.

Reverse Engineering

If it is necessary to reverse engineer an existing system to jumpstart the requirements definition, there are several possible ways to proceed. The easiest way to reverse engineer the requirements is the get a copy of the requirements document for the previous embedded system. Copy it and you are done.

This sounds good, and to the extent it is possible, should provide the first cut at the new system's requirements. Unfortunately it doesn't always work. There may not have been an original requirements document; or the feature set changed over the life of the product, and the requirements document may not have been updated.

If your need for accurate requirements is not met by the requirements document of a previous system, obtain the user's manual of the the previous system. This is often a good source of requirements data. It can be supplemented by actually using the previous system, and observing what it does.

If a previous requirements document, and the previous user's manual are unavailable (real engineers don't write documents), or don't give you enough information; attempt to find architecture and design documentation for the previous system. If you find such information, it may be sufficient to produce both use cases and a formal requirement list.

Read through all of the user, architecture,and design documents you have found, and attempt to create use cases first, and then the requirements list.

If there was no previous architecture and system documentation, or if it was inadequate to produce the use cases and/or the requirements; you must study the source code of the previous system. Even if the documentation allowed you to construct the use cases, it may not have been detailed enough to help you with the detailed requirements list. For that you may still need the source code.

One thing you can do with source code is to make call trees for every interrupt and every task you discover in the source code. Interrupts tend to handle functionality associated with the virtual machine layer. Tasks tend to handle functionality associated with the application layer. Carry the call trees down to the lowest level of functions that don't call any other functions. You will find useful details even at that level.

The source code and call trees should give you enough information to write the use cases and the detailed requirements list. If they do not, it could only be because the source code is so hard to read that you cannot decipher it. If that is the case, give up on reverse engineering and find another way to obtain requirements.

Requirements Workshop

When creating an entirely new system, or when you are unable to reverse engineer the previous system, it will be necessary to hold a series of meetings with persons who know how the new system must work. For many, the most enjoyable way to do this is with the requirements workshop.

The requirements workshop is a one or two day meeting (duration depends upon size of the new system), between the system developers and resource providers or stakeholders. There are two goals. The first is to introduce the stakeholders and developers who will be working together. The second goal is to familiarize the developers with the stakeholder's

requirements for the new system.

By working and eating together for a day or two, the participants will naturally get to know each other, and thus meet the first goal of the workshop. The second goal, transmission of the requirements, will take place during the work sessions.

The work sessions should include the following:

1) Introductions -- Introduce participants, lay ground rules, icebreaker exercise, agenda for the workshop.

2) Historical roots -- Answers the questions: How is the job of the new system currently done? This is a presentation by one or more system stakeholders. It will hopefully include a visit to an actual work site.

3) System operational environment -- Presentation of the environment in which the new system will operate. This should include the organizational environment, the physical environment, the regulatory environment, the sales environment, and the maintenance environment.

4) System development environment -- The developers give an overview of their own facilities, including a map to the location, phone numbers of key people, persons assigned to the project, their backgrounds, and key equipment that may be used on the project. The stakeholders describe the persons involved in their management, engineering, their background, and contact information.

5) System goals and constraints -- All participants cooperate in creating a prioritized list of each of the different things the new system must do or be, and each of the things the system must not do or be.

6) Future activities -- The participants agree who will produce use cases and a formal requirements list, and when they will meet to review them.

This workshop should be sufficient to jump-start cooperation between a team of developers and a group of stakeholders in a new embedded

system project. In case the developers and stakeholders work for the same company, the agenda can be somewhat abbreviated, especially if both groups are co-located.

See Chapter 10 of Managing Software Requirements: A Unified Approach[3] for a detailed discussion of the requirements workshop.

Requirements Model

Many developers, after they have the use cases and the requirements list, construct a so-called requirements model of the system. The requirements model is a first attempt to structure the application layer. It defines an interactive collection of virtual objects that meet the system requirements, and are, at least conceptually, capable of running on an idealized virtual machine layer.

Early requirements models used a data-flow virtual machine. Jacobson proposed a nifty, Interface-Entity-Control virtual machine for the requirements model.

While it helps to construct such a model, it is not always necessary.

If you do an architectural study like the one shown in the next section, you can document a collection of virtual objects that are compatible with the virtual machine and hardware layers of your target system.

Safety Issues

Embedded systems are frequently used in circumstances in which system failure endangers life or property. This is particularly true of military, and medical systems. It is also a consideration in automotive, laboratory, industrial, and consumer systems.

An important part of analysis is determining to what extent the proposed

system is hazardous to life or property, and what may be done to mitigate the hazards. Medical and military development environments have long employed standard procedures to build safety into embedded systems. The hazard analysis is one of the procedures used to focus attention on safety issues during analysis.

Create a list of all the hazards the new system may present to the developer, user, maintainer, or passersby. List the severity of each potential hazard, and give some idea of its probability of occurrance. The risk associated with each hazard is the product of its severity times it probability of occurrance. Suggest methods for mitigating the riskiest hazards, through provisions in the hardware, the software, or documented procedures for using the system. This may be the only safety-related thing you do during the analysis part of the development, but it will help to focus attention on safety issues early in the project. Hazard analysis is included in the requirements work because it may lead to additional requirements for the system. For more information on hazard analysis, start with https://en.wikipedia.org/wiki/Hazard_analysis

ARCHITECTURE

Architecture answers the question: "What are the major hardware and software components of the embedded system, and how do they work together to meet the system requirements? A variety of methods have been used to answer these questions. All work. Some are confusing. The method used here is easier than some, yet provides enough information, to move on into estimating and design.

In this sub-section, an **object oriented architecture** method is presented, using our example of the Theremin device. Then an **architecture workshop** is described, that will allow you to take advantage of possibly dispersed knowledge about the system architecture.

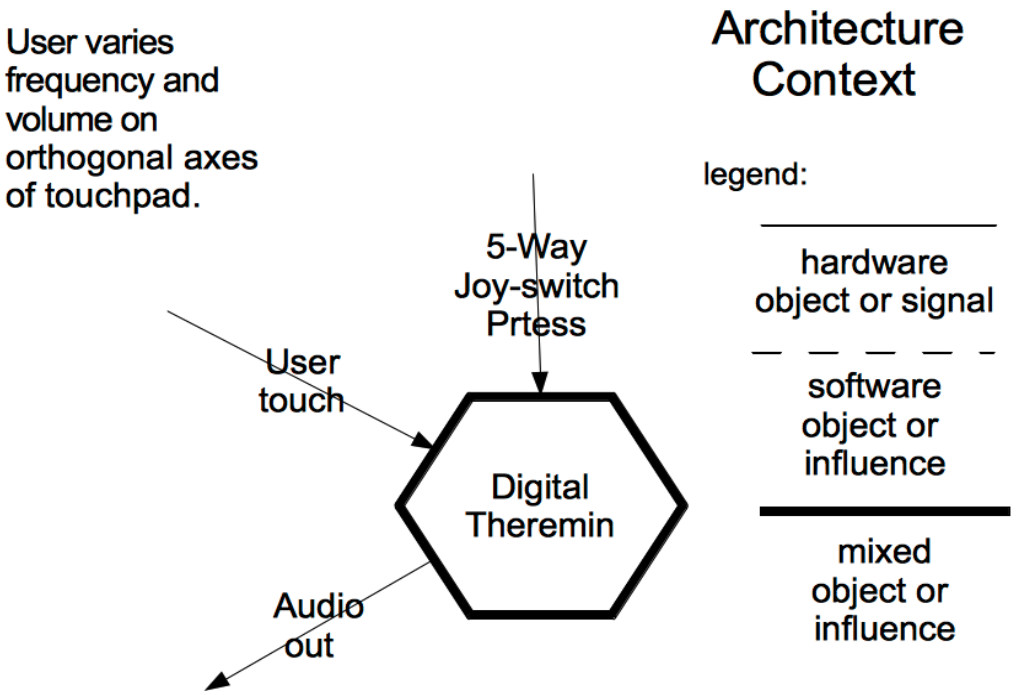
Object Oriented Architecture

The goal of an object-oriented architecture study is to describe a collection of hardware and virtual objects, which meets the requirements produced by analysis. This section shows what goes into an object-oriented architecture study, using the example of the digital Theremin.

Some definitions from the introduction are repeated here:

Objects may be hardware or virtual. A **hardware object** is made out of material parts. A **virtual object** is simulated in software. Objects may be composed of other objects. A **compound object** is an object which is composed of other objects. A **mixed object** is a compound object with some hardware and some virtual objects.

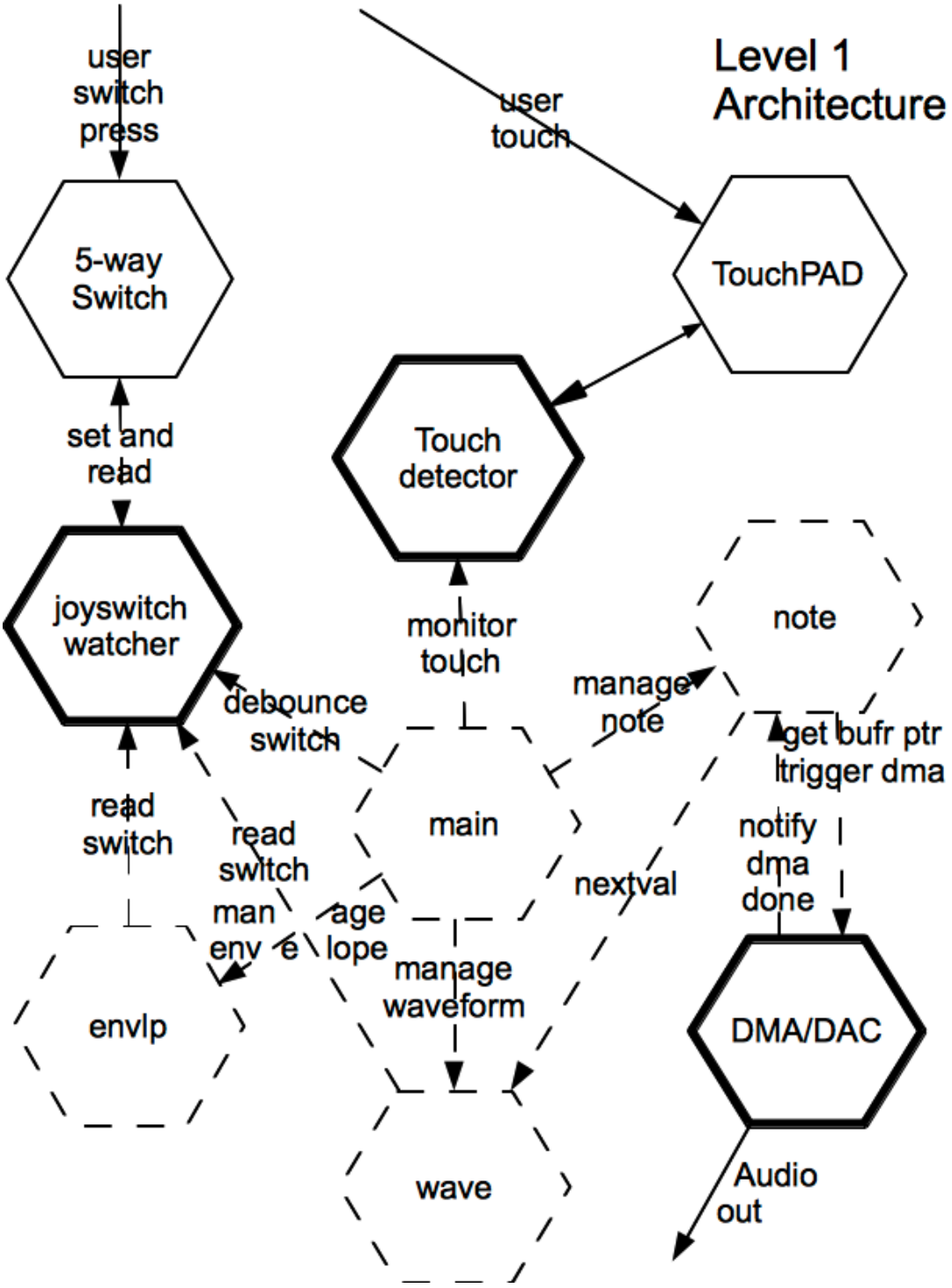
The architecture study begins with a single mixed object comprising the entire system. This object is shown along with its functionally relevant interactions with the external environment. This is called a system context diagram, and is shown on the next page.



The user touch controls the volume and frequency of musical notes. The joy-switch controls the waveform and envelope shape..

On the next page is shown a more detailed diagram of the system. It still includes all of the external influences on the system, but it breaks down the single mixed object, represented by the bold hexagon, into a group of objects, and shows the interactions between them. The breakdown is made along functional lines.

Virtual (software) objects and influences are shown with dotted lines. Hardware objects and physical influences are shown with thin, continuous lines. Mixed objects and influences are shown with bold, continuous lines.


Level 1
Architecture
user
switch
press
user
touch
5-way
Switch
TouchPAD
set and
read
Touch
detector
joyswitch
watcher
monitor
touch
note
debounce
switch
manage
note
get bufr ptr
trigger dma
read
switch
main
read
switch
notify
dma
done
nextval
man
age
env e
lope
manage
waveform
envlp
DMA/DAC
wave
Audio
out

TouchPAD
The TouchPAD is the actual touchscreen hardware device. It has a four wire interface which can be used by driver firmware or hardware to determine the location at which a single stylus is pressed against the screen.

Touch Detector
This mixed hardware/software object at the top of the screen gathers information on user touches and touch releases, including their locations, and responds to calls from the main loop to supply that information when requested. The Touch Detector is further decomposed in a Level 2 architecture diagram with the same name.

5-way Switch
The 5-way switch is a five position switch on the application board into which the LPC1768 board is plugged.

Joyswitch Watcher
The joyswitch watcher object is a mixed object whose debounced output is used to determine how to set waveform and envelope shape. The Joyswitch watcher is further decomposed in a Leevel 2 architecture diagram with the same name.

Main module
The main object receives control from them mbed startup code immediately after power-up. It initializes the reset of the application modules, and then enters the main loop, which repeatedly debounces the touch screen and joy-switch objects and calls update methods for the wave, envlp, and note objexts.

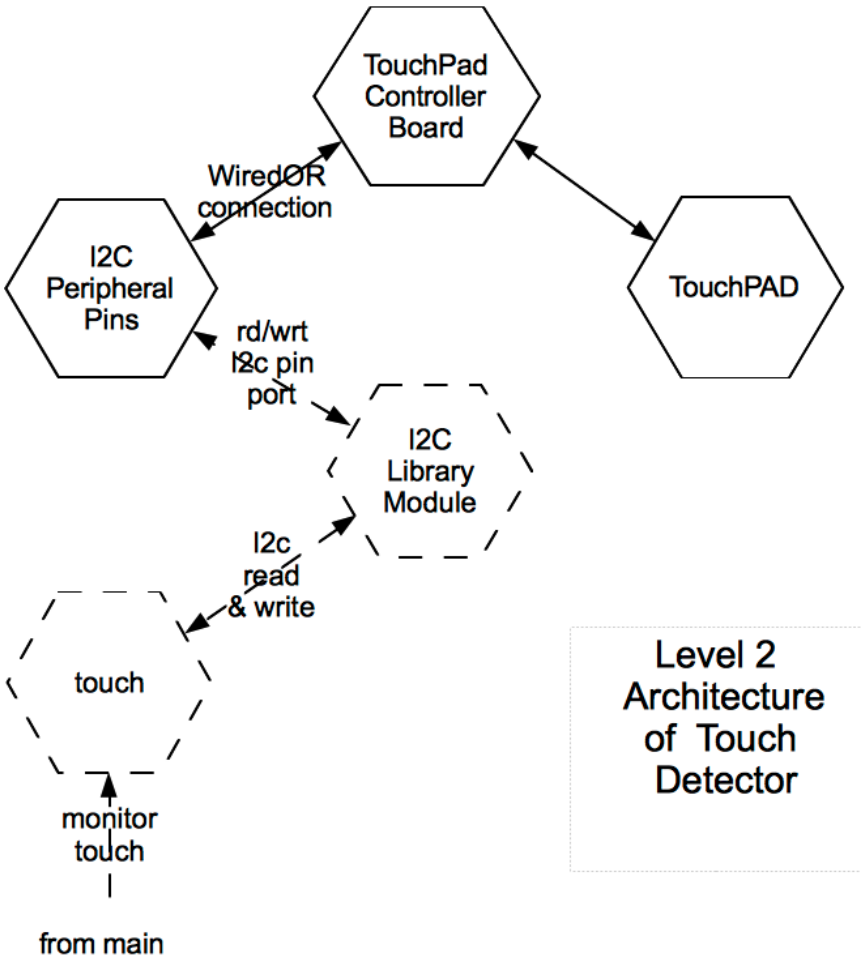
Wave module
The wave module computes each buffer that is passed to the DMA based upon the supplied frequency, the most recent choice of waveform, and wave phase computed for the last DMA buffer.

Note
This module triggers the production of notes, manages the envelope of the notes, sets instantaneousl note volume and bends the note pitch in response to touch screen input, and fills each buffer requested by the dma object.

Envlp
The envelope module is a collection of functions used by the Note module to manage the factor which modulates the waveform produced by the Wave module

DMA/DAC
This module initializes the DMA and DAC hardware, disables the DMA when requested, notifies the note module a DMA buffer iwhen it is time to fill another DMA buffer, and prepares a recently filled buffer for transmission by the DMA. The DAC/DMA module is further decomposed in a level 2 architecture diagram shown on a subsequent page.

> Note: Hexagons are used for objects in this book because they allow for closer packing of objects in diagrams and provide more pathways for influences passing between the objects.

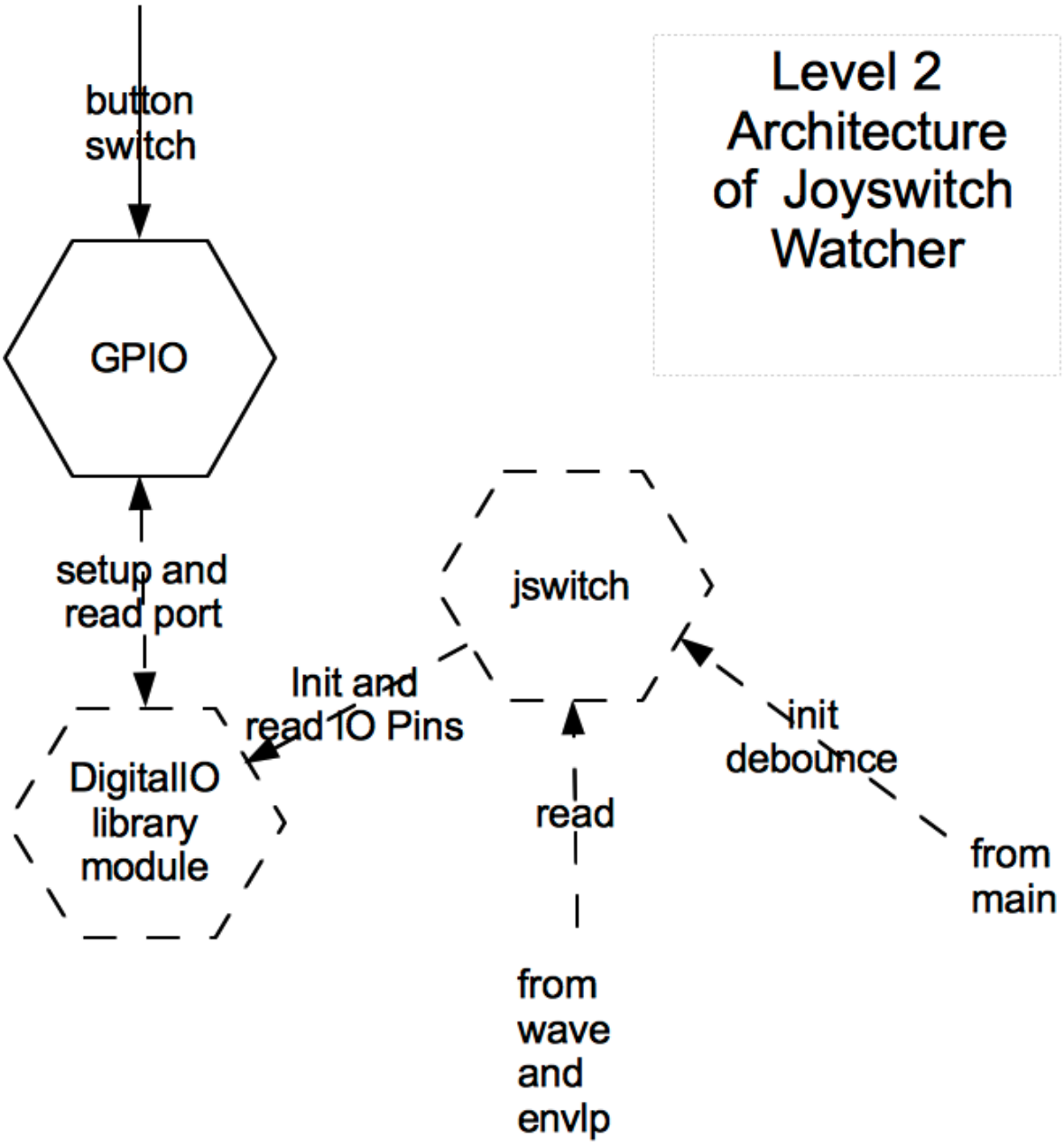


TouchPad Controller Board

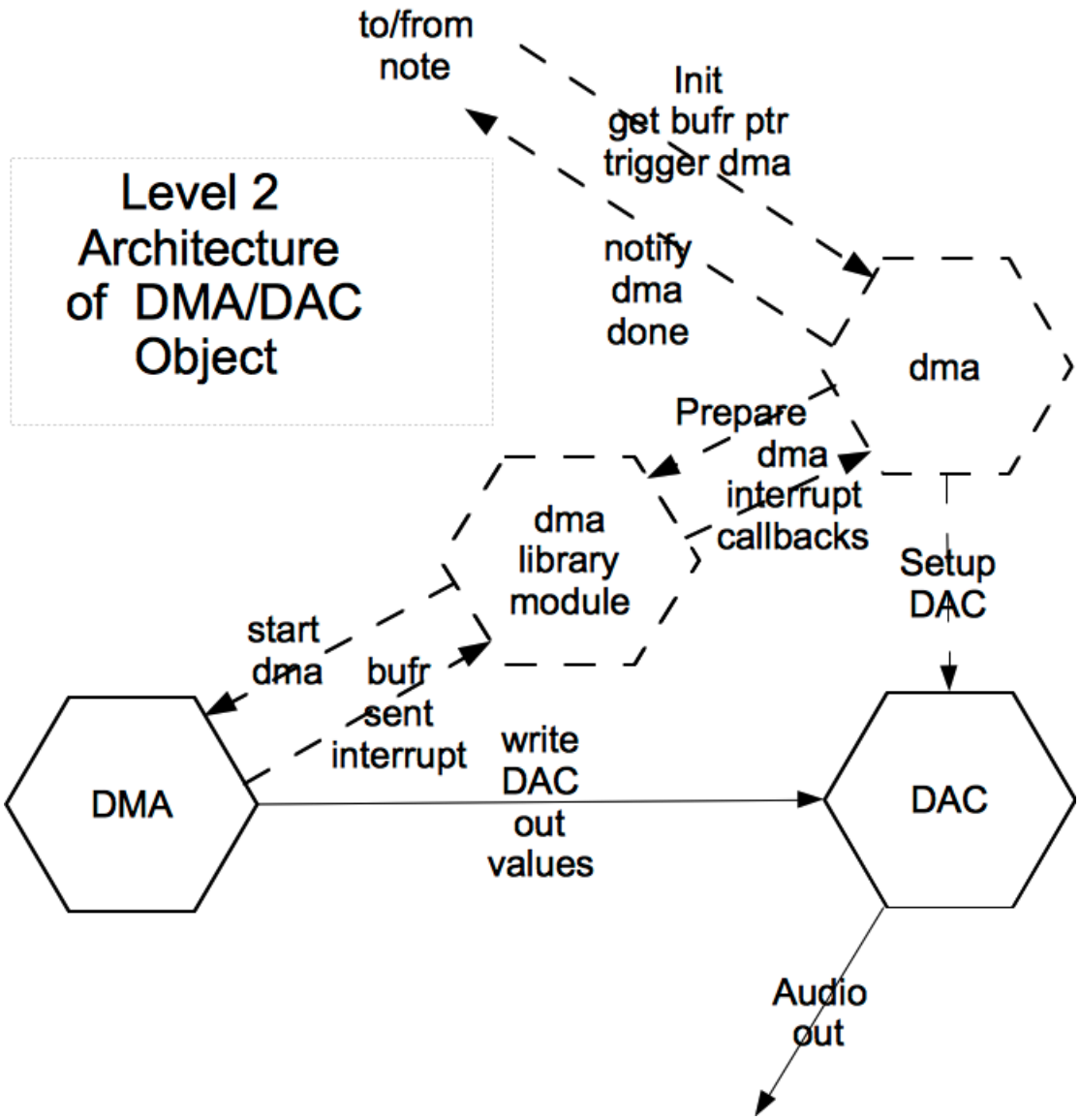
This is the board with the STMPE610 resistive touch screen controller on board. The touch driver module communicates with the board over an I2C link via the mbed i2c library module.

Touch

The application module that monitors the touch pad via the i2c library and pins is called touch.



The wave and envlp modules use the application module called jswitch which debounces the button joyswitch using the Digital IO library.



The dma module sets up the DAC with a register write, and tells the DMA channels to output to the DAC. After each interrupt it notifies the note object to refill the buffer that just completed, and sets up a new dma transfer with the Prepare() method of the dma library object.

This completes the architecture diagrams of the digital Theremin project. These architecture diagrams can be modified in the design phase to give detailled information about the interaction of the application virtual objects.

Architecture Workshop

It may sound strange, but a group of electrical, mechanical, and software engineers will typically enjoy a morning spent functionally decomposing an embedded system.

I have seen groups of engineers take over an architectural workshop, and produce dozens of diagrams of the different functional aspects and levels of an embedded system... and enjoy it.

Engineers like clarity. An architectural workshop brings clarity to their understanding of the relationships between the components of the embedded system.

It's easy to start an architecture workshop. You just make sure that the engineers show up on the morning in question at the appointed spot. You bring a flip chart and some markers. You introduce them to the concepts of hardware objects, virtual objects, compound and mixed objects, and you lead them through the creation of the system context diagram.

Next you help them with the Level 1 diagram. By the time they have finished level 1, the engineers are ready to take over, each of them leading the decomposition of a piece of the system that appears on the Level 1 diagram.
(It's a good idea to have worked out in your own mind what the Level 1 diagram should look like before you help others produce it.)

By the time everyone runs out of gas, you have two things: a whole bunch of diagrams of the embedded system architecture, and enhanced clarity in the minds of everyone present. Take the diagrams and put them into an architecture document. The clarity will pay off in everybody's work for the rest of the project.

ESTIMATING

Estimating is a skill that only grows with practice. Always do a project estimate when the architecture is done. Do this whether you are asked to or not. Over time your methods and results will improve.

Produce your estimate using three different techniques. I use the **Cocomo**, **Modified Function Point**, and **Add Up the Guesses** techniques.

Once you have your three estimates, combine them in whatever way your conscience, your bank account, and your creativity dictate.

Don't worry about being wrong. You probably will be, but if you give it your best effort; you will not suffer unduly when your estimate is later proved incorrect.

Of course, if you have your own money riding on the outcome, it helps to be right.

COCOMO

First you guess the number of lines of code in your project. Then you make swags at the values of a collection of parameters which describe the project. Finally, you apply the COCOMO formulae to compute the total effort, calendar time, and number of programmers necessary to accomplish the mission. For details, refer to the book: Software Cost Estimation with COCOMO II[4].

For our digital Theremin example, the modules to be coded are main, wave, note, envlp, touch pad driver, and dma. Only part of the dma module has to be written, as there is a DMA example program available on the mbed.org website.

Lines of code guesses for the modules to be written are:

main	50
wave	65
jswitch	25
note	200
envlp	100
touch pad driver	200
dma	100

I guessed 740 lines of code, not counting libraries already written by mbed, and for parameters, picked:

parameters	**setting**
mode	organic
Analyst capability	high
Application experience	high
complexity	nominal
Database Size	low
Language experience	high
Programmer capability	high
Required reliability	nominal
Requirements changes	nominal
Schedule pressures	low
Main storage constraint	low
Execution time constraint	high
Use of Software Tools	high
Computer Turnaround time	low
Virtual Machine Experience	nominal
Virtual Machine Volatility	nominal

Executing the magic formulae returned the following values:

Total Man months	1.9
Total Calendar months	3.19
Developers Required	0.6

Modified Function Point Method

In the normal function point method, you add up the number of inputs, outputs, transaction types, intermediate file types, external file types, and so on to arrive at the number of function points.

For embedded systems, you can just add up the number of influences and objects in your architectural diagrams. If you don't have any architecture diagrams, make some. Use that count as the number of function points. Then you apply a collection of formulae to obtain a variety of results. For the digital Theremin device example, the number of function points came out to be 39. Using formulae in a script program sent to me (thanks Kevin), the estimate came out to:

Function Points	58
Paper pages deliverable	107
Document words deliverable	42658
Function Pt Growth when done	5.2%
Critical creep	14.6 %
Test Cases	130
Defect Potential	160
Defects Shipped	4.5
Build Staffing	0.39
Maintenance Sgtaff	0.11
Years of Use	2.8
Total Effort	1.96 man months

The substitution of object/influence count for function points was arbitrary; but the results are in pretty good agreement with other estimates. It's probably best not to admit to anyone that you are using modified estimation methods, which are not supported in the literature. If you want to use function point analysis in a more formal way, see the book Function Point Analysis, by Garmus and Herron[5].

Add Up The Guesses

In this method, you write down everything you are going to do and guess how long each action will take., for example:

Task	hours
Figure out why I'm doing this	1
Do use cases	1
Do architecture diagrams	8
Estimate project effort	4
Research algorithms	4
Initial Class (Module) diagram	4
Steady State Interaction Diagram	4
Initialization Interaction Diagram	1
Module Descriptions	16
Set up development environment	16
Write hardware and tool test program	6
Setup IDE and makefile	10
Code and debug main module	12
Code and debug jswitch module	8
Code and debug wave module	8
Code and debug envlp module	8
Code and debug touch module	24
Code and debug dma module	12
Integrate, test, and revise	40
Devise tests to verify algorithms	16
Devise test to demonstrate it works	4

Total Hours	207
Man Months	1.24
Calendar Months = Man Months * 2	2.48

(The factor of 2 in Calendar months because I expect to work only half-time on the project.)

Combining The Estimates

A weighted average works OK here. Then the problem of combining the estimates reduces to one of choosing the weights. You weigh more heavily those estimates that conform to your gut feeling about the project. Then you find some convenient way to rationalize that decision. It is perfectly OK, and may be wise, for the weights to add up to a number substantially bigger than 1.
Just looking at the three estimates and guessing, it looks like the digital Theremin device is going to take somewhere in the neighborhood of 2 calendar months to complete. It will require anywhere from a third to half of my time during those 2 months.

Appendix A contains cellular calculator scripts for the COCOMOII and Function Point calculations.

TOOL SELECTION

For the digital Theremin project, the mbed.org website provided quite a choice of compilers. I chose the Code Sourcery free arm_none_eabi gnu compiler toolset. It installed quickly on both Linux and Mac OS X, and was compatible with the IDE called SlickEdit, which is superior in some ways to EMACS, but costs a lot more.

With ARM processors, compiler suppliers usually provide libraries for the processor, and for many of the available eval boards. Fortunately, mbed.org provides its own processor and board libraries as good or better than anything supplied by compiler makers.

Since there is not much parallel processing going on in the digital Theremin, there is no need for an RTOS. Even when there are several parallel processes, their activities can often be better coordinated by devoting a state machine to each process, rather than incurring the overhead and interrupt latency of an RTOS.

There is no JTAG connector on the processor board, so a JTAG debugger is out of the question. Code can be loaded by dragging the compiled executable file to a mass storage device that appears on the desktop whenever the processor board is connected to the PC via USB.

The USB connection includes the ability to direct printed output from the running program to a remote terminal program running on the PC. So most debugging can be done by printing information to the terminal on the PC.

A logic analyser was required to debug the I2C connection between the mbed board and the touch screen controller board.
The Saleae USB Logic Analyzer available from SparkFun is the best availabe 8-bit logic analyzer, and it worked nicely for the digital Theremin project.

HARDWARE SUPPORT

Often, a electrical engineer develops a printed circuit board for the while you are writing requirements and doing the architecture. A mechanical engineer may simultaneously develops the mechanical components.

The hardware engineers can be helpful with things like getting the right power supply for the prototype boards, making sure the processor is running, providing standoffs for the circuit board to keep it off of the desktop, assembling and disassembling cases, and learning the history of the product.

In return, they will expect you to provide early versions of software to test board features, light blinking programs to test visibility of LEDs, and sometimes to help probe mystifying problems with the circuit board.

The project is not going anywhere until you get production printed circuit boards. Since there is such a long lead time on circuit boards, you must test as many of the board features as possible on the prototypes, as early as you can. This is where you learn how the board actually works, which is vital to finishing the development of the virtual machine object designs.

A good relationship with the hardware engineers is not only vital to the success of the project. It is vital to your own success. Fortunately, most hardware engineers are pretty easy to get along with. They have had most of their rough edges rubbed off by contact with reality. There are notable exceptions to that rule.

In the digital Theremin project, the processor, application, and touch screen controller boards were purchased off the shelf. There was no need to support debugging those boards. However it was necessary to configure and make the connections between the touch screen and the processor and controller boards.

DESIGN

The architecture phase identified virtual objects and described, in a general way, how those objects interact to accomplish the use cases described in the requirements.

Design picks up where architecture leaves off. Its purpose is to describe a collection of virtual objects and their methods and data elements, which work together to meet the system requirements.

> Note: The design techniques used in this example draw heavily on the modelling methods of the Unified Modelling Language inspired in part by Grady Booch[2], created by Rational and supported by IBM. That language was intended to support objected-oriented software technology using classes, objects, inheritance, and polymorphism in method invocation.
>
> The typical embedded system has no more than one to three objects per class, usually one, and no need for inheritance. In such an environment, objects and classes are practically equivalent. For that reason, the design techniques can be simplified to leave out class and inheritance, and focus only on the objects of the design.

The documentary output of the design process is a collection object interaction diagrams, an object overview diagram, and an object catalog. The object interaction diagrams are similar to the diagrams of the same name in The Unified Modeling Language User Guide[1].

The object overview diagram is an abbreviated graphical representation of the objects, data elements, and methods of the design.

The object catalog is a more detailed, textual elaboration of the overview diagram.

When the design is done, the object catalog may be broken into chunks and handed out with the diagrams to different development team members. The diagrams help the team members develop their own mental concrptual map of the entire system. The catalog defines the scope of their

contribution to the overall effort.

Note: For small systems, with only one software developer, in the absence of regulatory requirements for design documentation, it is possible to let the code itself serve the purpose of the object catalog.

The table below gives a 3 step guide to producing the documents, and hence the design.

For each use case do the following:

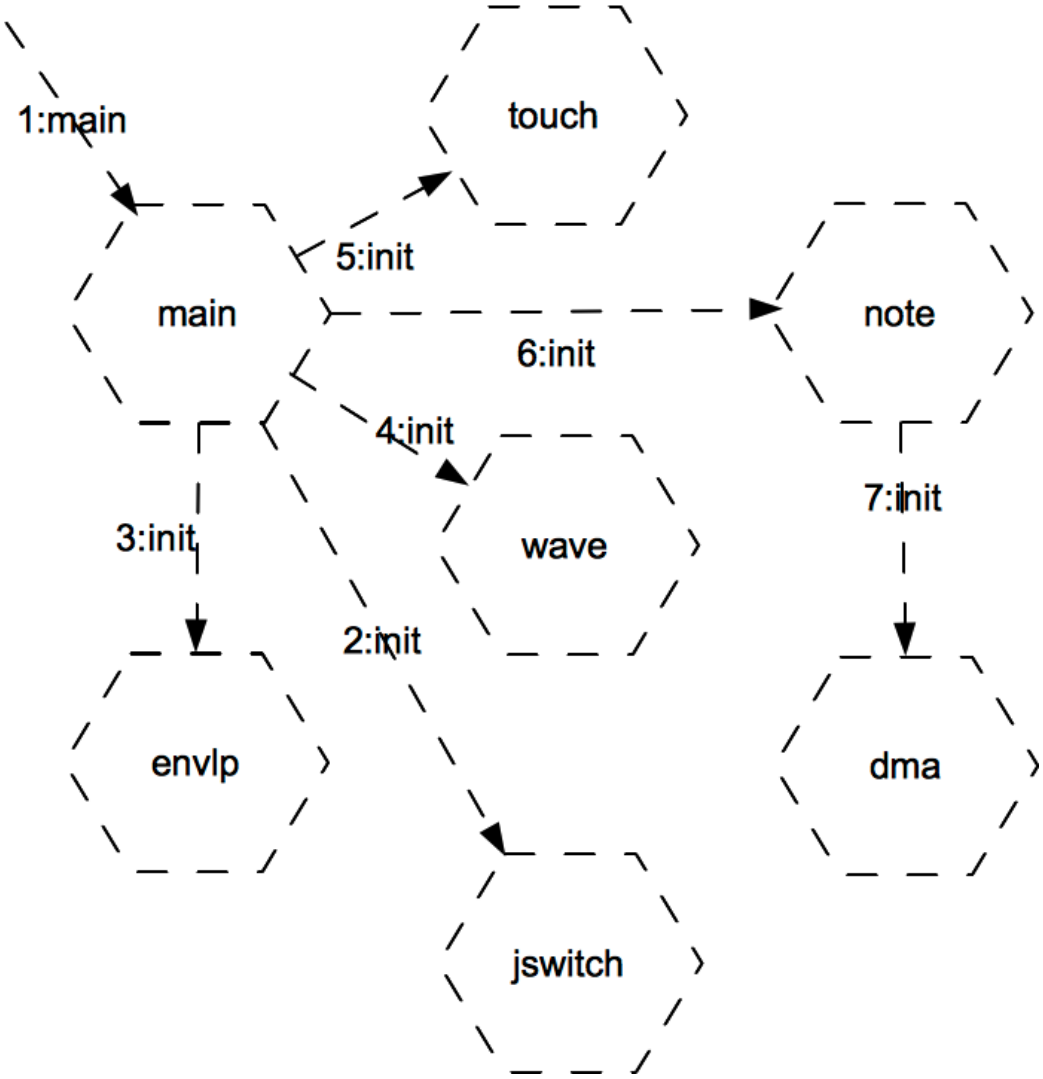
Step 1	Make a list of virtual objects from the architecture that will be needed for this use case. If something is missing, revise the architecture. For each virtual object in the list, make up short descriptive names of the object, and it's methods and data elements that will be needed to support the use case.
Step 2	Draw a diagram showing the virtual objects for this use case, and connect the objects with arrow-tipped lines representing the methods of each object. For each virtual object in the diagram, work out in your head or write down in a notepad what each method should do. If the same method is used in previously diagrammed use cases, resolve any differences in the description, or define a new method or revise the architecture and design as necessary.
Step 3	If the designer is satisfied with the virtual objects and methods for this use case, add them to the object catalog. Otherwise, repeat steps 1 and 2.

The next few pages record the outcome of the above three step design loop for the digital Theremin.

Object Interaction Diagrams

Whatever the embedded system does, it is a good idea to start design work with its drive train. This is the object interaction set that produces the primary motivated activity of the system. For the digital Theremin, the drive train is the collection of objects and methods that support the use case PlayTheremin.

Nevertheless, we begin here with the initialization use case, since it offers the opportunity to introduce the object interaction diagram with a very simple example.



Object Interaction Diagram for PowerUp Use Case

Each of the objects actually used in the power-up use case is shown as a dotted hexagon, representing the object. The dotted arrows are method calls made from one object to another.

Each dotted arrow is annotated with a number followed by a colon, followed by the name of the method called in the target object.

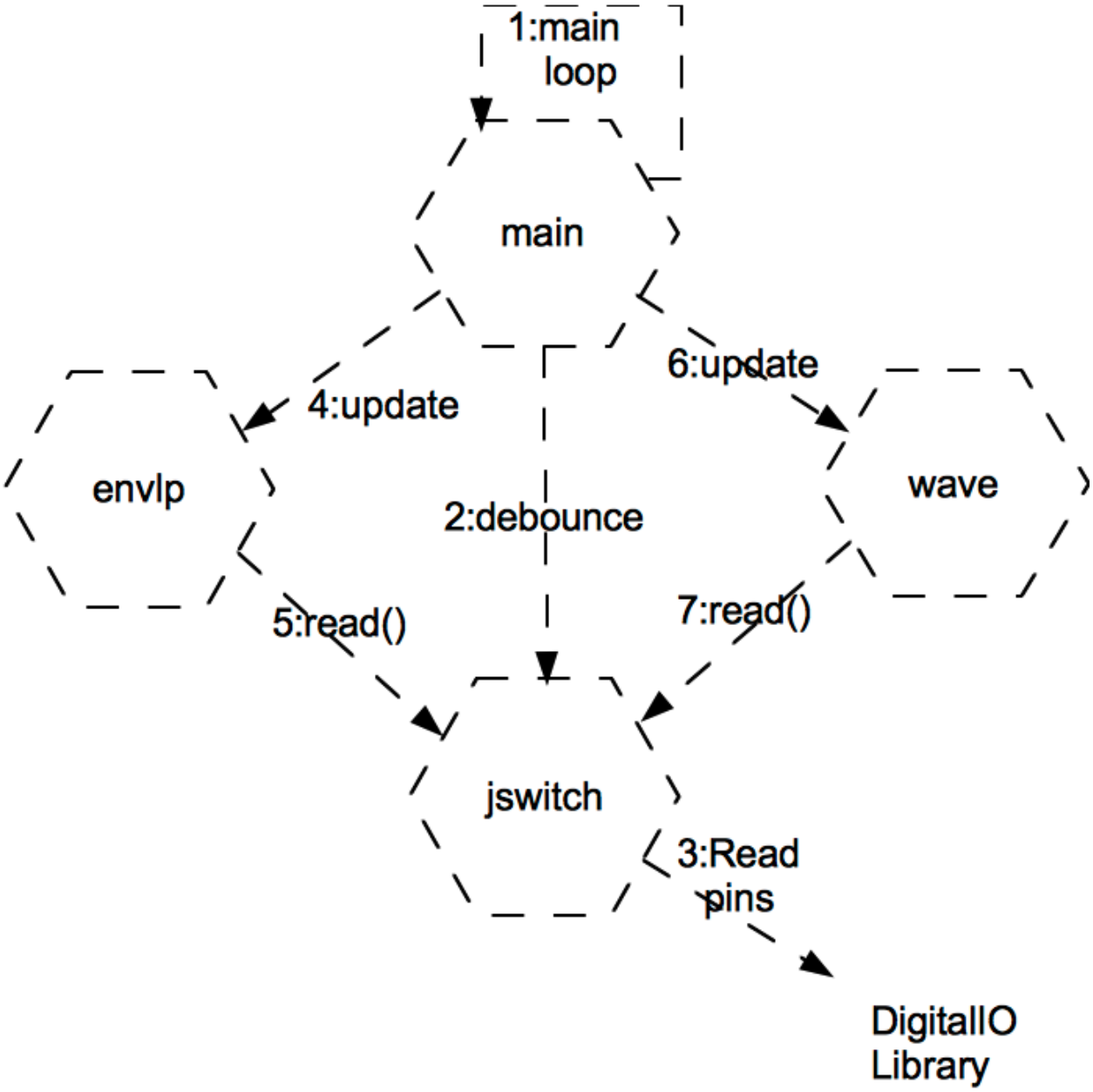
After the chip power up code, the first virtual object call (1) goes to the main() method of the main object.

The main() method then calls the init() methods of the jswitch, envlp, wave, touch, and note objects in that order.

The note object calls the init() method of the dma object.

That is how an object interaction diagram shows the methods called and the order in which they are called.

Next, we will look at the Set Parameters object interaction diagram.



Object Interaction Diagram for Set Parameters Use Case

All of the method calls in the Set Parameters use case are made under control of the main loop, labeled 1.

The main loop first calls the debounce method (2) of the jswitch object. This method reads five digital IO pins (3) with methods named after each pin. Then it combines all of the readings into a single bit field and checks

for any persistent bits set. Bits that remain set through four or more calls are Or'd into the official switch setting bit field.

Next the main loop calls the update method (4) of the envlp object, whichcalls read (5) to look for bits in the Up, Down, Left, or Right bits of the official bit field. If an Up or Down bit is set, update increments or decrements, respectively, a variable that tells how many buffers to use in the attack phase of the envelope.

If a Left or Right bit is set, update increments or decrements, respectively, a variable that controls the duration of the release phase.

Then the update (4) method then clears all the bits that were set in the official switch setting bit field.

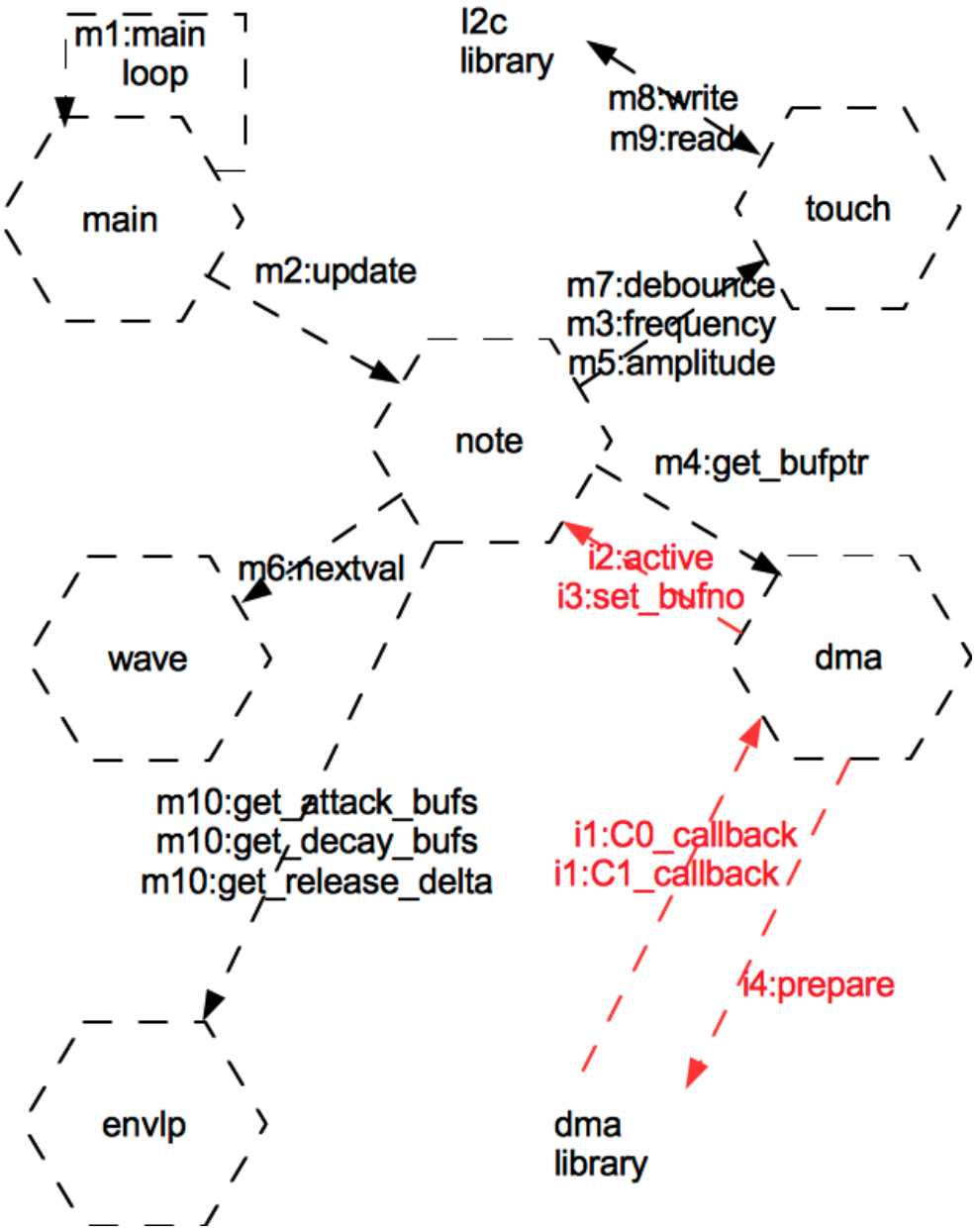
After update (4) returns to the main loop, the main loop calls update (6) of the wave object.

Like the envlp object, the wave object calls the read (7) method of the jswitch object to find out is the Center bit of the official bit settings field is set.

If the Center bit is set, the wave update (6) method sets the note waveform for the next note to the next waveform in a circular list of waveform templates kept in the wave object.

Then it returns to the main loop. The next time a note is started, the new settings for attack and release duration and waveform will be used.

The Object Interaction Diagram for the Play Theremin use case was saved for last, because it is the most complicated.



Object Interaction Diagram of Play Theremin Use Case

Some of the method calls in the Play Theremin use case take place under

the control of the main loop. Those method calls prefix an 'm' to their sequence number, and are shown in black.

Some of the method calls take place under the control of the DMA interrupt in the dma library code. Those method calls prefix an 'i' to their sequence number, and are shown in red. The interrupt routine in the DMA library is activated whenever the DMA completes a buffer transfer to the audio output DAC.

Only two DMA buffers are needed. The buffers are filled and output alternately by the note object. When buffer 0 completes, C0_callback (i1) is called by the dma interrupt. When buffer 1 completes, C1_callback (i1) is callec by the dma interrupt.

Each of the callbackl routines checks to see if there is a note in progress by calling the note active (i2) routine. If so the callback routine calls the note method set_bufno (i3) to tell the note object which dma buffer to fill. Then the callback routine calls the dma library prepare routine to begin output of the next (already filled) buffer.

If there is no note active, the callback routine disables the DMA channel in use.

Each time through, the main loop (m1) calls the note update method (m2).

The note update method first calls the internal state_machine method to get the current frequency (m3) and manage the transitions between note phases.

Then it checks to see if the dma interrupt has used the note set_bufno method (i3) to tell it to begin filling the next dma buffer.

If a new buffer has been requested, the note object calls the dma get_bufptr method (m4) to get a pointer to the requested buffer, and then calls its internal fill_buf method to fill the requested buffer.

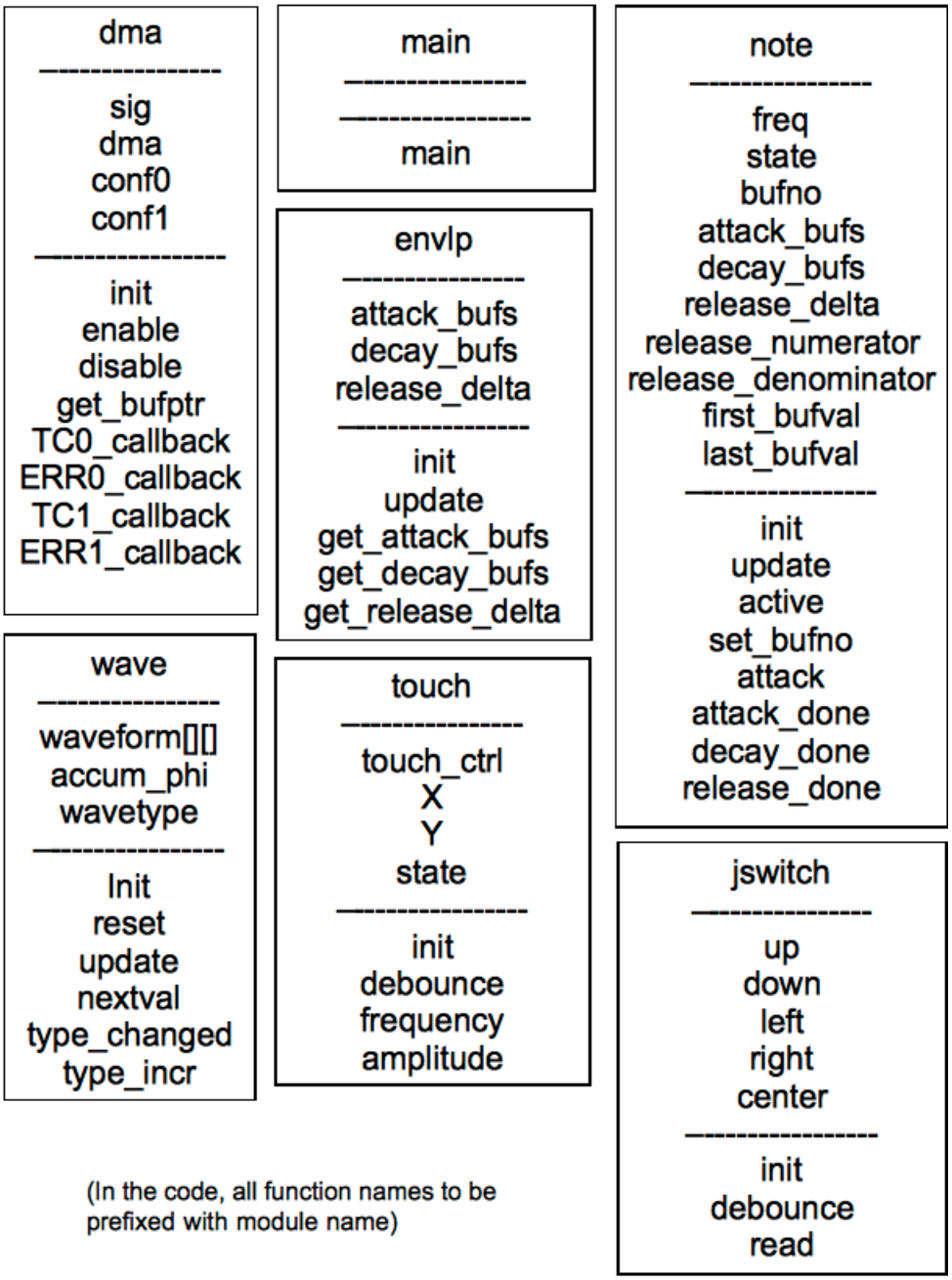
The fill_buf method calls the touch object's amplitude method (m5), and then enters a loop with as many iterations as there are waveform values in a dma buffer. Each time through the loop, it calls the wave nextval method (m6) to get the waveform value which is multiplied by the amplitude to get the value to place in the next index position in the dma buffer.

Once the next dma buffer is filled, the fill_buf routine increments a buffer_counter for the envelope phase it is working on, and returns to the note update method (m2).

The note update method (m2) then calls the touch debounce method (m7), which uses the i2c library methods write (m8) and read (m9) to update the touch/no touch indication, and the X and Y values if a touch is in progress.

After that the note update method (m2) decides whether it is time to start a new note or release the current note, if any. If it is time to start a new note, the note update method calls envlp methods (m10) to get the envelope parameters for the new note.

Object Overview Diagram



OBJECT CATALOG

The Object Catalog for the Digital Theremin describes each virutal object to be coded, its encapsulated data structures, its public methods, and significant private methods. Each object corresponds to one or more source modules. The purpose of the catalog is to supply enough information to code the modules associated with each object.

There's no right amount of information to put into the object catalog. If the programmer is also the designer, or if the project is small, the object catalog will probably contain less information than it would contain for a larger project. The object catalog template below suggests some information to supply for each object.

Object name: <object name>

<Brief description of what the object does.>

Source modules: <all source modules for object>

Data elements:

Name	*Type*	*Purpose*

Public and significant private methods:

Name	*Return Type*	*Arguments*	*Purpose*

Diagrams, algorithms, etc.

< Any material needed to help code the object>

Suggestions for verification:

<Suggestions for verifying the operation of the object>

Object name: main.cpp

Initializes the digital Theremin code and enters loop calling the note update method.

Source modules: main.cpp

Data elements:

Name	*Type*	*Purpose*
none	n/a	n/a

Public and significant private methods:

Name	*Return Type*	*Arguments*	*Purpose*
main	none	none	Initialize digital Theremin code and enter infinite loop calling each object's update method.

Object name: touch

The touch object uses the i2c library code to access the SMTPE610 touchscreen controller for the purpose of reading presence or absence of a user touch, and X and Y values of that touch.

Source modules: touch.cpp, touch.h

Data elements:

Nate	*Type*	*Purpose*
touch_ctrl	I2c interface object	Provide i2c access to the touchscreen controller
X	short	X coordinate of a touch
Y	short	Y coordinate of a touch
state	int	state variable for debounce state machine

Public and significant private methods:

Name	*Return Type*	*Arguments*	*Purpose*
init	none	none	Initialize touchscreen controller
debounce	boolean	none	Debounce touch indicator returned by controller, return true if touch present, false otherwise.
frequency	int	none	Return frequency corresponding to touch X value.
amplitude	int	none	Return amplitude corresponding to touch Y value.

Suggestions for verification::

1. Add debug code to show the X,Y readings of a touch. Touch all four corners of the screen. Write down the X,Y coordinates of each corner of the screen.

2. Verify that the max amplitude returned by touch_amplitude() is within 16 counts of 255, and that the minimum amplitude returned by touch_amplitude() is less than 16.

3. Verify that that the max frequency returned ty touch_frequency() is within 20 Hz of 1047.

4. Verify that the minimum frequency returned by touch_frequency() is within 10 Hz of 44 H

Object name: note

This object manages the production of a note in response to a user touch on the touch screen. It also handles setting the attack and release parameters if the white button is down when the touch happens.

Source modules: note.cpp

Data elements:

Name	*Type*	*Purpose*
freq	int	Current frequency of note in progress, if any
bufno	int	Index of next buffer to fill, set by dma object call to set_bufno. Reset to NIL on completion of note fill_buf() routine, which is called by note update() method
state	enum	Current state of note state machine. One of ATTACK, DECAY, SUSTAIN, RELEASE.
release_numerator	int	Numerator of fraction to multipy amplitude by for each successive dma buffer in the release phase.
release_denominator		Denominator of fraction to multiplyi amplitude by for each successive dma buffer in release phase
attack_bufs	int	Number of dma buffers to use for attack phase of note
decay_bufs	int	Number of dma buffers to use for decay phase of note

release_delta	int	Integer to subtract from release numerator for each new dma buffer.
first_bufval	int	First value in next dma buffer
last_bufval	int	Last value in next dma buffer

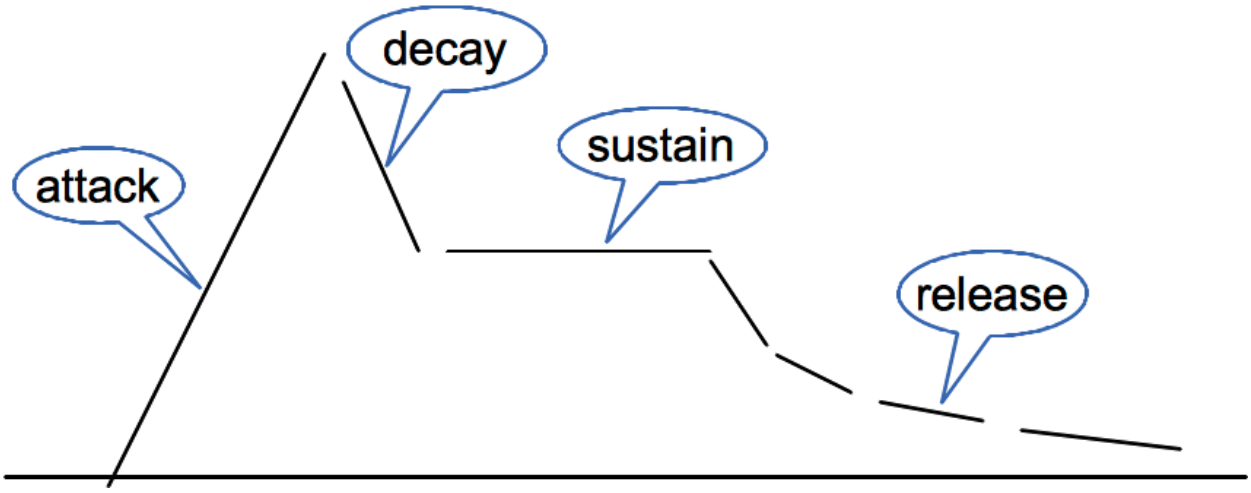
Public and significant private methods:

Name	*Return Type*	*Arguments*	*Purpose*
init	none	none	Initialize dma, envlp, and wave objects. Sets starting values for data elements.
update	none	none	Calls note state machine to manage waveform envelope. If bufno is not NIL, call fill_buf() to fill the next dma buffer. Calls touch_debounce() to find out if there is currently a user touch. Starts or releases a note as needed.
active	bool	none	Used by the dma object to determine whether a note is currently active.
set_bufno	none	int	Used by dma object to tell note object which of the two dma buffers to fill.

attack	none	none	Starts a new note by setting some variables and entering the NOTE_ATTACK state of the note state machine.
attack_done	bool	none	Called by the note state machine in NOTE_ATTACK state to determine if it is time to enter the NOTE_DECAY state.
decay_done	bool	none	Called by the note state machine in NOTE_DECAY state to determine if it is time to enter the NOTE_SUSTAIN state.
release_done	none	none	Called by the note state machine in NOTE_SUSTAIN state to determine whether it is time to go to the NOTE_OFF state.

fill_buf	none	none	Gets a pointer to the next dma buffer. Computes the envelope values for the first and last sample in the dma buffer, depending upon note state and last known amplitude. Enters a loop computing values for each sample in the dma buffer, depending upon first and last sample values computed above. Updates envelope depending upon note state. This method is not public but is included because it is a key method in the system

Diagrams, algorithms, etc.



Typical note envelope, showing the four stages of a note. The attack phase starts when the user touches the touchscreen. Decay phase happens after the prescribed number of attack phase dma buffers have been sent. The sustain phase begins after the prescribed number of decay phase dma buffers have been sent. The release phase begins when the user removes the touch. The release state is over when the amplitude drops below a threshold value.

Suggestions for verification::

1. Hook up an oscilloscope to the audio output jack. Verify that notes which are not interrupted before the end of the release phase go through the four phases shown in the diagram.

2. Verify that notes which succeed each other in quick succession, without a complete release phase, start by cutting off the preceeding note, and then beginning the attack phase.

3. Verify that the note amplitude increases as the X value of the touch screen increases, and that the frequency of the note increases as the Y value from the touch screen increases.

4. Verify that when the white button on the application board iis pushed up repeatedly, the note attack becomes softer, and the duration of the attack phase is longer as seen on an oscilloscope.

5. Verify that when the white button on the application board iis pushed down repeatedly, the note attack becomes harder, and the duration of the attack phase is shorter as seen on an oscilloscope.

6. Verify that when the whitre button on the application board is pushed left repeatedly, that the duration of the release phase decreases as seen on an oscilloscope.

7. Verify that when the white button on the application board is pushed right repeatedly, that the duration of the release phase increases as seen on an oscilloscope.

Object name: wave

The wave object has templates for each possible waveform (just three in this example). It maintains the current waveform phase, and returns the waveform template value for that phase on request. The phase is updated by an amount proportional to the note frequency divided by the sample rate each time a value is returned.

Source modules: wave.cpp

Data elements:

Name	*Type*	*Purpose*
wavetype	int	Tells what waveform template to use
waveform	Int [][]	Waveform template. One array for each waveform.
accum_phi	unsigned	Accumulated phase

Public and significant private methods:

Name	*Return Type*	*Arguments*	*Purpose*
init	none	none	Computes the waveform templates, and zeros the accumulated phase
nextval	int	unsigned	Computes the next sample value depending upon the accumulated phase, and frequency supplied by its argument

Diagrams, algorithms, etc.

The waveform buffer contains DMA_BUFSIZE samples. One cycle of the waveform is represented in the buffer.

The phase variable , accum_phi varies from 0 to 99,999, representing phase angles from 0 to 2*Pi radians. The change in phase from one sample to the next is:

delta_phase = 100000 * frequency/SAMPLE_RATE

The next value of accum_phi is:

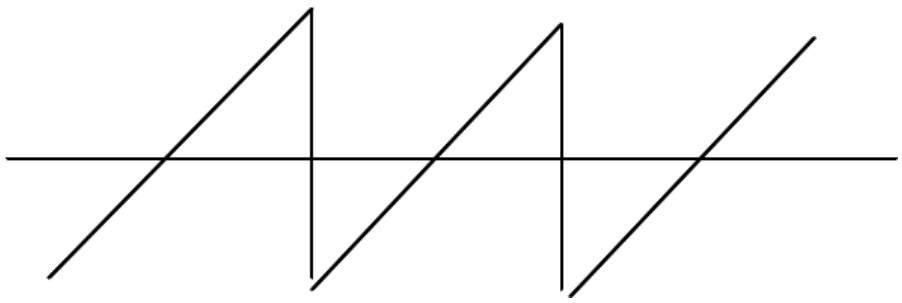
accum_phi = (accum_phi + delta_phase) % 100000

The index of the next value to return from nextval() is computed according to the formula:

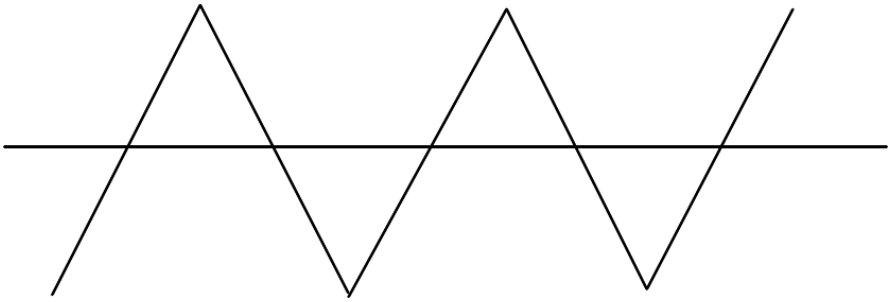
Index = (DMA_BUFSIZE -1) * accum_phi / 100000

Suggestions for verification::

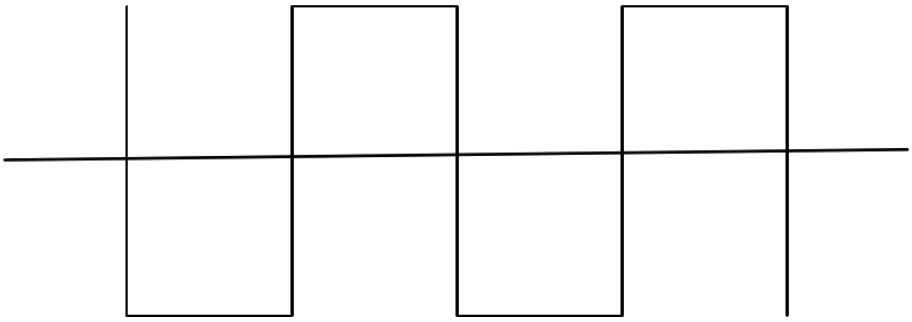
1. Hook an oscilloscope up to the audio output jack. Verify that immediately after powerup the output is a sawtooth waveform, as shown below



2. Verify with the oscilloscope that the sawtooth waveform changes to a triangle waveform the first time the white button on the application button is pressed inward in its center position.



3. Verify with the oscilloscope that the triangle waveform changes to a square waveform the next time the white button on the application button is pressed inward in its center position.



4. Verify with the oscilloscope that the square waveform changes back to a sawtooth waveform the next time the white button on the application button is pressed inward in its center

5. Verify with the oscilloscope that the minimum frequency that can be output using the touchscreen is within 10Hz of 43 Hz.

6. Verify with the oscilloscope that the maximum frequency that can be output using the touchscreen is within 20 Hz of 1047 Hz.

Object name: envlp

The envlp object is a collection of methods that the note object uses to keep track of the envelope value, a factor in computing the sample values sent by the dma to the DAC output pin. The envelope value changes in a piecewise linear fashion, with each piece being one dma buffer.

Source modules: envlp.cpp, envlp.h

Data elements:

Name	*Type*	*Purpose*
attack_bufs	int	Number dma buffers in the attack phase of the note.
decay_bufs	int	Number of dma buffers in the decay phase of the note
release_delta	int	A value to subtract from the maximum envelope value to get the release_numerator.

Public and significant private methods:

Name	*Return Type*	*Arguments*	*Purpose*
init	none	none	Set initial values for variables.
update	none	none	Call the jswitch read routine to decide how to set values for attack_bufs, decay_bufs, and release_delta
get_attack_bufs	int	none	Return the current value of attack_bufs
get_decay_bufs	int	none	Return current value of decay_bufs

get_release_delta	bool	none	Return current value of release_delta
set_release_delta	void	Int	Used by note to control release duration

Diagrams, algorithms, etc.

The envelop is composed of piecewise linear segments, each of which contains one dma buffer worth of samples. There is an approximation made to simplify computation of the envelope at dma buffer boundaries. The first sampe of a dma buffer is assumed to be equal to the last sample of the previous dma buffer. The theory is that this approximation will not result in undue distortion.

A better approach might be for the first sample of a new dma buffer to be computed by the previous fill_buffer routine, in the same way the last sample of that buffer was computed, and saved for the next dma buffer.

Suggestions for verification::

1. Hook an oscilloscope up to the audio output jack. Verify that the output waveform does not develop discontinuities (other than the expected one at 2*pi radians) as the envelope amplitude changes

Object name: jswitch

This objects watches the five digital inputs of the joystick-switch, and debounces and latches any of the five possible switch closures. It allows clients to select switches for checking, and if latched, clearihg the closure indications.

Source modules: jswitch.cpp, jswitch.s

Data elements:

Name	*Type*	*Purpose*
official	uchar	Contains a bit set for each unread switch closure.
up	DigitalIO	Read the up switch
down	DigitalIO	Reads the down switch
left	DigitalIO	Reads the left switch
right	DigitalIO	Reads the right switch
center	DigitalIO	Reads the center switch

Public or significant private methods:

Name	*Return Type*	*Arguments*	*Purpose*
init	none	none	Initialize debounce and report algorithm
debounce	none	none	Debounce and latch switch presses
read	uchar	uchar	Read adn clear any bits specified by the mask argument

Diagrams, algorithms, etc.

Keep an official variable containing unread switch closures. Clear all bits that are read using the mask of the read routine.

Suggestions for verification::

Use debug facility to print the value of official every time it changes. Press some joystick buttons and verify the behavior of the variable called official.

Object name: dma

This object initializes the dma configuration, enables and disables the dma, receives the dma interrupts, and uses them to request the note object to fill the next dma buffer.

Source modules: dma.cpp

Data elements:

Name	*Type*	*Purpose*
sig	AnalogOutput object for DAC	Dma output destination
dma	MODDMA library object	Object which keeps configuration of the dma channel
conf0	MODDMA ptr	Points to dma configuration object for buffer 0
conf1	MODDMA ptr	Points to dma configuration object for buffer 1
buffer	Int[][]	Dma buffers 0 and 1

Public and significant private methods:

Name	*Return Type*	*Arguments*	*Purpose*
init	none	none	Initialize dma configuration objects for buffer0 and buffer1

enable	none	none	Prepare configuration for the first buffer and setup the DAC
disable	none	none	Disable both dma channels
get_bufptr	Int *	int	Return a pointer to the dma buffer whose index is supplied.
TC0_callback	void	void	Called by dma interrupt when first dma buffer transfer completes. Disables the dma channel for the first dma buffer. If a note is active, calls note objects set_bufno routine and re-enables the dma channel for the second dma buffer. Otherwise disables the second dma channel as well
ERR0_callback	void	void	Called by the dma interrupt when the first dma buffer transfer fails. Issues an error message to anybody listening on stdout.
TC1_callback	void	void	Called by dma interrupt when second dma buffer transfer completes. Disables the dma channel for the second dma buffer. If a note is active, calls note objects set_bufno routine and re-enables the dma channel for the first dma buffer. Otherwise disables the first dma channel as well
ERR1_callback	void	void	Called by the dma interrupt when the second dma buffer transfer fails. Issues an error message to anybody listening on stdout.

Diagrams, algorithms, etc.

The sample rate chosen is 22050 Hz. There are 512 samples per dma buffer, so the dma outputs slightly more than 43 buffers per second. Since the template waveform is the same size as the dma buffer, the minimum frrequency that can be output is 43 Hz.

Suggestions for verification::

1. Plug earphones into the output jack of the application board. Listen to the output at various note frequencies. Verify that there is no significant 43 Hz flutter in the audio output heard.

2. Examine the audio output with a spectrum analyzer. Verify that the 43 Hz component is always a lot lower in level than the primary frequency played.

Design In Safety

Embedded systems are frequently used in circumstances where system failure endangers life or property. This is particularly true of military, and medical systems. It is also a consideration in automotive, laboratory, industrial, and consumer systems.

In the section on Analysis, the Hazard Analysis was mentioned as a tool to investigate the hazards presented by the new system. In the design portion of the project, other tools may be used to promote safety in the end product. These tools are the Failure Mode Effects and Criticality Analysis and the Fault Tree Analysis.

Faillure Mode Effects Analysis (FMEA)

This is a formal technique for examinine the effects of hardware or software faults in components of the system, and assessing their potential for creating safety hazards. This technique is best applied during the design

portion of the project, when some knowledge has been developed of the components, both real and virtual of the system. For more information, search the web for "Failure Mode Effects Analysis".

Fault Tree Analysis

The FMECA looks at safety from a bottom-up view. If this breaks, what hazards might that present. The fault tree looks at it from the top-down. Starting with the list of potential hazards, you ask: "What has to go wrong for this to happen?". Then you design in software or hardware changes to prevent that fault from happening. For more information, search the web for "Fault Tree Analysis".

These tools will help you decide where the greatest risks are in the system, and how you might effectively counter those risks. More information on the subject of sofware safety is available by searching the web with the keywords "Software Safety".

Design Review

Design review is a long tradition in the engineering profession. It carries a bit of emotional charge because engineers are putting their creative work up for critical review by their peers. This can be a productive activity, that points up shortcomings in the design; or it can be a useless exercise in ego stroking, or coworker bashing.

On balance, the design review is worth doing. It often results in an improved product. The best way to approach the design review is to keep it as low key as possible. The designer should invite those people he thinks most likely to contribute. The review should take place in a series of small meetings over a period of days, or even a couple of weeks.

Only those persons who are in a position to contribute to the technical process should attend. All others, especially upper management and financial stakeholders, should stay away, if that is possible. They are apt to

misunderstand engineering discussions.

Never make a design review a critical milestone related to payment of development funds. That will cause the review to be rushed, and carried out in a possibly adversarial manner.

Make sure the reviewers have read the documents before coming to the meetings. If they have not, cancel the meeting.

Take notes on any action items that result from the design review, and put those notes in the project archive. Depending upon the kind of process used by your organization, the notes may be needed in a project audit.

CODE

If the analysis and design activities are done thoroughly, writing the code is an easy task. Nevertheless, preparation for coding sometimes plunges the development team into minor conflicts. No two software engineers write code in exactly the same way. Each engineer has his own preferences and cognitive style. The three most likely conflicts are: Choice of Language, Choice of Operating System, and Coding Standards.

Choice of Language

Most embedded systems are written in 'C', but there are alternatives. C++ are thought by some to provide better tools for translating object-oriented design concepts into source modules. Java and C# are also desirable languages, partly because of the marketing efforrts of Sun Microsystems and Microsoft; but also due to their large, useful libraries.

The author's view is that C++ is better suited to large systems having many objects of each class, than it is to embedded systems. Some of the desirable features of C++, such as exception processing, and virtual functions, come with increased code size, or performance limitation.

Java and C# still have issues with interpreters and garbage collection that might preclude their use when interrupt latency is a major consideration.

Object-orientation is far more important in the design phase than when writing code. Once the code is compiled, there is no important difference between code written in C++ and code written in C. Furthermore, the need for writing strictly typesafe code in an object-oriented language often complexifies implementations involving function tables and registration-callback patterns.

If you are able to use 'C' in your embedded project, you will seldom go wrong in doing so. If there is significant pressure to use another language, such as C++ or C# you may have to go along. If that happens, take advantage of the unfortunate choice to learn all you can about

implementing useful design patterns in those languages. It's sometimes a challange, and almost always interesting.

It should be noted that a choice of C++ may be made without incurring any significant performance or code size disadvantage. One has only to insure that the features of exception processing, and polymorphism are not used.

In fact, you can take a perfectly good C program and convert it to C++ by simply changing the source module suffixes from .c to .cpp. That may be a good way to overcome a knee-jerk insistance on C++.

Choice of Operating System

Many embedded systems applications involve multiple parallel processes. There might, for example, be one process which operates to a communications protocol, one which manages a robotic arm, and one process which interacts with a user over a simple keypad.

PC Operating System

The mainframe and PC solution for such situations is a multi-tasking operating system. Each process is given its own logical task, implemented by function calls from a for or while loop. The communication between tasks is implemented with semaphores, critical sections, pipes, or even network protocols.

The use of a PC operating system has more to recommend it than just handling multi-tasking. It allows the developer to use common PC components for the user interface, and network communication. This can save a bundle in engineering labor re-inventing common display and networking components, both hardware and software.

The downside of the PC OS approach shows itself when the developer needs to write a driver for custom hardware, so it can be used with the PC. The cost of implementing drivers for PC operating systems can be a lot more than one might expect

When using a PC operating system there is another problem as well. The user interface component will often be written iin an interpreted language that is different from the language needed for any custom robotic components of the system. This forces the developer to connect the user interface to the hardware drivers through a command language and comm protocol.

Configuration control of the Linux operating system is a particular problem, since it is continually undergoing changes from different people and organizations separated by geographical and cultural differences.

Use of the Windows operating system exposes one to Microsoft tendency to use technology features to softly coerce increased use of Microsoft products in the organization using the embedded system.

Rtos

A special class of operating systems called real-time operating systems have been implemented to mitigate the code size, and interrupt-latency issues common with PC operating systems.

The specially designed RTOS's often require considerable use of scarce system resources, such as processor time and memory. Sometimes even the RTOS's interrupt latency is too much for some poorly designed circuit boards. The RTOS also forces the use of elaborate threadsafe interprocess communication tools, which might not otherwise be necessary.

State Machines

An alternative to the use of a PC operating system or RTOS is the use of a main loop which calls separate state machines for each parallel process.

This technique requires the parallel processes to be divided into small chunks, each of which is implemented as a separate state of the process state machine. Each time a state machine is called by the main loop, only one of its states executes. When that chunk of the state machine finishes,

it transfers control to another chunk or "state", which is called the next time through the main loop.

The state machine approach substantially reduces overall code size compared to an RTOS implementation. It also simplifies interprocess communication, since one chunk is never interrupted by another. And it reduces interrupt latency, since there is no operating system code which must be protected by disabling interrupts. There will, of course, remain thread safety issues related to communication with interrupt routines.

Coding Standards

A British philosopher once observed that there are really only two kinds of people in the world: the simple-minded, and the muddle-headed. On software development teams, that dichotomy often manifests as those who are sticklers for the appearance of source code, and those who prefer broad lattitude for personal expression.

The sticklers want the look of source code to be tightly controlled. The manner of nesting braces must conform rigidly to a preferred scheme. Capitalization or non-capitalization of variable and function names must be subjected to rigorous collections of rules. All flow of control statements must use braces, whether they need them or not.

The broad brush ones don't appreciate nitpicking of their artistic deployment of coding symbols. They usually accept coding standards that affect the object code, but purely stylistic issues are seen as invasion of their personal perogative.

This conflict is probably a reflection of different cognitive styles. Persons in the two groups simply use their brains in different ways.

Fortunately, there are tools to resolve this issue without creating conflict. If you are a member of a team that has chosen a coding standard that doesn't fit your worldview, purchase an editor that supports multiple styles of pretty-printing. There are several such editors, and most support the

generally favored coding styles of sticklers. Then you can write your code any way you like, but you use the editor to reformat it before checking it into the project repository.

In our digital Theremin code, C++ is the chosen language. That choice was due to the necessity of using mbed library components written in C++. The application objects, nonethless, make the minimum possible use of C++ features; and so their code looks much like C code.

The only two processes involved in the digital Theremin code are the main loop and the dma interrupt, so no operating system or collection of state machines was needed.

Appendix B contains the source code for the principal application objects of the digital Theremin. The library code is available through mbed.org.

DEBUG

Debugging presents a continuing series of riddles to the developer. Each riddle is a consuming mystery, right up to the point when it is discovered to result from an obvious mistake.

Coding, Debugging, and Integrating often merge into a single confusing process. Here we consider debugging to be the work done on individual objects to make sure they perform according to their published interfaces (public method specifications). This work is usually done by the person writing the code, with a little help from his friends, if any.

Bug Categories

Most bugs fit into one or more cateories. It helps to bear in mind these categories during debug and integration. They serve as hypotheses to account for the deep mysteries of your bugs. Below is a list of common embedded system bug catagories, some of which occurred in the digital Theremin project. The list is not exhaustive. In fact, it covers only a fraction of the problems you will encounter. You can find other bug lists on the internet, or in books devoted to software testing.

One-off errors

This usually consists of starting or stopping a loop one interation too soon or one iteration too late. It can easily result in memory corruption or buffer overflows (see below).

Buffer overflow errors

Continuing to copy data beyond the end of buffer causes memory corruption errors (see below).

Type casting errors

Casting one kind of variable to another is always a suspicious activity. That's why Lint pays such close attention to improper type casting. It is a

particularly good candidate if you have a bug in an arithmetic algorithm.

Careless syntax errors

We all make this kind of mistake. It is hard to catch without the help of other team members.

Poor thread safety

Accessing the same variable from a lower and a higher priority thread should be done only when exclusive control of the variable can be guaranteed.

Bad pointer errors

This leads to memory corruption errors and a variety of other conditions arising from reading unexpected values.

Variable name errors

Make sure that your variables have the names you think they do. If you mispelled a variable name, and it happened to correspond to the name of another variable in the system, you've got problems.

Unhandled arithmetic exceptions

Divide by zero errors are the most common glitches of this type. Unnoticed overflow is another. Floating point routines are particularly susceptible to this problem.

Inadequate design

This usually results from failure to consider all possibilities in an algorithm, a data structure, or an object interaction. That, in turn, often happens when there is a rush to write code, at the expense of design effort.

Pipeline errors

In pipelined processors, instructions are not always executed in the order they are read by the processor, particularly when interrupts are processed. I once had a pipeline-related error that prevented interrupts from being disabled, even though the disable instruction itself was clearly being fetched.

Stack overflow

An easy bug to fix, this fault can produce a bewildering variety of symptoms. This ought to be an early suspect in memory corruption, and weird execution sequence bugs. Just add more stack and see if that fixes or delays the problem.

Memory allocation errors

Freeing unallocated memory, freeing the same memory twice, and overwriting memory accounting fields have long troubled systems with dynamic memory allocation. Even if you don't have a heap, you may still have allocation errors in buffer managers, pipe managers, and linked list node arrays.

Misunderstanding the hardware

This includes a variety of problems, such as setting up chip select signals incorrectly, setting the clock to the wrong speed, and failing to specify the correct number of wait states for blocks of memory or I/O space. On processors supplied with a variety of powerful hardware peripherals such as flexible serial ports, DMA's, and timer modules, it is easy to botch the setup a desired hardware configuration. Some processors have different modes of arithmetic or addressing operation that must be carefully set and monitored.

Careless I/O port assignment

Getting the port mask wrong for a signal means you are not looking at the signal you thought you were seeing. Alternatively, you are not driving the

signal you thought you were driving.

Operator precedence errors

Parentheses take care of most of these errors, though not all. In the example below, a wicked memory corruption bug arose from the sequence:

```
if(k<N) {
    x[ k ] = a + x[ k++ ]; break;

} else continue;
```

The problem here is that the post incrementation takes place after the limit check on k, but before the assignment. The result is a variable one word past the end of x[ ] is modified by the assignment.

Variable scope errors

These can be largely avoided by avoiding the same names in automatic, static, and global variables. It is also handy to use an object prefix for all the variables in an object, or to make all such variables members of a structure that contains all the object variables.

Link editing errors

It is a good idea to verify that memory actually exists at every location where a variable is placed by the linker, and that it is the type of memory that you were expecting for that variable.

Variable alignment errors

Some processing operations only work when variables begin on a long word boundary. Some compilers align variables which are structure members on byte, word, or doubleword boundaries depending upon compiler switch settings. Some DSP buffers must be aligned on boundaries having a number of trailing binary zeros equal to the that in the next power of 2 greater than or equal to the buffer size. Failure to comply with all such

constraints creates memory corruption bugs.

Algorithmic errors

Are you sure you implemented that algorithm correctly?

Memory corruption errors

Memory corruption can manifest in a bewildering variety of ways. You may be overwriting any pointer or variable or piece of RAM-resident code in the system. The fault may not become apparent for many seconds, when it is too late to trace it to its source. Deduction and logic analyzers are good tools for attacking memory corruption errors. Checksumming sections of code or data can also help.

Timing errors

You need a good multi-channel oscilloscope or a logic analyzer to investigate timing bugs. Find or create signals which show all of the timing relationships in the problem area. Then look at those signals on the scope to make sure things happen at the rate, and in the order expected. Oftentimes, just putting the problem on the scope is enough to find the cause.

Initialization errors

Every compiler and linker outputs memory reservations for something called a BSS area. That is the Binary Storage Section. It contains the locations for all of those variables which are not constants or statically initialized.

Sometimes the C startup code will zero the BSS area, but sometimes it will not. The upshot is this: always specifically initialize every variable used in a program. Initialize that variable both statically and dynamically. The static initialization should be good enough unless your system has a warm-start capability, which permits restarting the main program without going through the complete boot-up process. Then you also need the dynamic

initialization.

Warm start errors

Warm start refers to a situation in which the code is restarted without powering down the system. Since the system is not powered down, RAM memory contents are not destroyed. Often the compiler initialization code is not run, or is not run in the same way. Warmstarting is frequently parameterized with an error code left over from the previous run of the system, or some other remnant of a previous run.

Every different way of warmstarting the system offers a completely new set of initialization errors to explore. Each powerup and parameterized warmstart must be understood in complete detail as to sequence of activities, and expected variable values.

If at all possible, avoid warmstarting the system all together.

Circuit board errors

Circuit designers make mistakes too. When they do, you may think you are reading a serial clock signal when you are actually looking at the ground plane.

The only way to catch these errors is to watch the related signals on the scope to verify that they are working as expected. If they are not, get data sheets for the related chips, and trace the malfunction through the circuit board. Better yet, demonstrate the problem to a hardware engineer, and let him trace the circuit malfunction.

Programmable logic errors

These are like circuit board errors except you cannot trace the signals beyond the pins of the gate array or programmed logic device. For that you need the equations which were used to program the device. If you find a pin on a programmed logic device that is not producing the desired signal, check first the inputs which are supposed to produce that resulting signal. If

they are right, either turn the problem over to a hardware engineer, or consider changing careers and becoming a hardware engineer.

Compiler errors

In thirty years of embedded systems work, I have discovered an average of one compiler bug per system. That's not to say that there weren't other compiler bugs in each system. I just didn't catch them. For any given bug, there is a small but finite chance it results from the compiler not behaving as expected. Thus, it is always a good idea to look at how the compiler has translated suspect code, to see if this might be that single compiler bug for the project. Don't do this first, though, or even twenty first. This should be the first of the so-called desperation checks.

Library errors

Modern processors are often accompanied by sample library code which exercises most of the features. Evaluation boards also come with more extensive libraries which implement common things such as USB drivers, mass storage drivers, and internet protocols. More often than not, you will have to find hidden bugs in one or two of the library routines you use.

Spurious interrupt errors

Sometimes even hardware designers forget to do things like tie signals to power or ground. If those signals happen to be hooked up to processor interrupt pins, you can get some pretty mystifying failure modes. Catch these bugs early by writing a spurious interrupt routine that is reached from every interrupt the processor accepts, that is not used for a system interrupt. Make sure that the spurious interrupt routine makes it perfectly clear what is wrong.

Power supply noise

Avoid if possible, working on sensitive electronic instruments which use on-board switching power supplies. If you cannot avoid this situation, be on the

lookout for peculiar spikes in signals read by the software. Nothing messes up a simple signal processing algorithm faster than rhythmic spiking on its input signal lines.

Advice for Debuggers

For an embedded software developer, speculating on hardware causes for bugs is evidence of desparation in the search for the bug. It may very well be a hardware bug. Several of those crop up during the course of a project; but the dominant probability is that the software is at fault.

When another developer tells you he thinks your software has a bug in it, respond immediately. He may be solving your problem for you.

When you have looked for the same bug for a long time, your mind gets stuck. You will not find the bug until you alter your mental frame of reference. Do whatever you normally do to accomplish that feat. This can mean leaving the workplace, or engaging in a variety of stress-reduction behaviors, including sleep.

Describe your bug to a team member. For 20% of all bugs this results in the immediate solution of the problem.

Never make major changes to source code late in a debugging session. You will almost always find yourself restoring the code to its earlier state, if you saved a backup copy.

Fear and debugging are incompatible. If you are afraid for your job, you will not find the bug. Fear is either due to your own feelings of unworthiness, or to a climate of fear in the workplace. If it is the former, get help. If it is the latter, find another job.

When you are totally stumped, and you tried standing on your head and that didn't help, consider each of the bug catagories above for applicability. Get a book that contains common causes of bugs, such as Software Testing Techniques[7]. Find other bug catagory lists on the internet. You will

seldom find a difficult bug without first forming a hypothesis as to its cause.

Never blame the development environment, the tools, or another team member for the bug you are tracking. The bug is in the software, not in the environment, not in the tools, and not in your co-workers. Improve your circumstances if you can, but find the bug regardless of your circumstances.

Don't let debugging get you down. You could be one of those people who have to work for a living.

INTEGRATE

During integration, the hardware and virtual objects created by the developers are brought together and made to work with one another in accordance with the system design

With a decent system design, competent work by the developers, adequate debugging tools, and the absence of schedule pressure, integration can be an exciting and rewarding experience. The more the above conditions are not met, the deeper the rung of hell on which you will find yourself during integration.

Integration is an extension of debugging. You are still looking for bugs, but now they are harder to find because you need the help of all of the other team members. There is usually some anxiety about whether the dang thing will work at all. This anxiety is especially common among resource providers, who by this time, have stuck their neck out a mile and a half supplying the money for the project.

The keys to a successful integration are a good design, a cooperative attitude, and optimism. The better the system design, the fewer problems will crop up in integration. The better each team member's attitude, the more likely the team is to pull together. The less likely it is to indulge in blaming and finger pointing.

The surprising thing about integration is that, in this phase, optimism is an asset. In analysis, design, coding, and debugging, the engineer focuses mostly on what can go wrong. Pessimism is a vital part of the mindset necessary to deal with physical reality. In the integration phase, the precautions have all been taken. You are now looking for everything to fit together and work. A little bit of optimism at this point can help you persist through whatever difficulties arise.

Advice For Integrators

Integrate early and integrate often. The earlier in the project you can build

up an end-to-end version of the system (even if most parts are stubbed out), the less confusion you will encounter when everybody is adding in their own contribution. Build an early skeleton of the system. Add to it regularly as code becomes available. If target hardware is not available, integrate modules on PCs, when that is feasible. Simulate hardware components that are unavailable.

Assign one team member the role of point-man for integration. This person will make the first attempts at an end-to-end system, and will provide the initial release that is used by other team members in the early stages of integration.

There will be many different configurations of the software under development and test simultaneously during integration. Pick a configuration management tool and stick with it. Develop procedures for defining releases, and for tracking the uniquely versioned source modules in each release.

Always keep a baseline release of the system. This is the release that includes working versions of the most virtual objects, without any special debugging additions or enhancements. This release represents your latest and greatest working system. It is the release you will deliver when it finally contains working modules for every object in the system.

Archive copies of the baseline release once a day. That way the most you can lose is a day's work. Assign someone the job of updating the release archive, and of knowing roughly what is in each archived release.

Public finger-pointing or blaming other team members is counter-productive. If you discover a problem with another team member's contribution, explain privately to that person what you found, and ask for their help in resolving the issue. Do not bring it up in a public meeting unless it affects others besides you and the other responsible team member.

Never assume you are not the cause of someone else's problem. Keep up to date on the problems other team members may be having. Spend some time each day asking yourself how you might be responsible for those problems. Try to help the other team members solve their problems even if you are convinced they are not related to your own work.

When you are stumped on a problem, seek help from other team members. Ask the whole team to listen to your description of a problem, and suggest ways to find its cause. Integration does not happen without communication.

VERIFY

Verification is a more or less formal way of checking your work. It is like proofreading a document. Verification catches the mistakes you didn't see when you were debugging or integrating.

In projects where safety is an issue, verification tends to be more formal. It might include the creation of overall verification plans, the definition of detailed testing protocols, and publishing of detailed reports of every test, its outcome, whether retesting is necessary, and so on. It always includes detailed testing of any code added to the system to mitigate hazards.

In the digital Theremin project, suggestions for verification were included in the Object Catalog part of the design. This section contains the verification report, which lists the tests performed on each object, and the outcomes of those tests.

The short block of text below is a template to use in documenting the verification tests for each object.

Object: <name of object to test>
Test Number: <number of test for this object>

Test Description:

<description>

Test Outcome:

<Outcome>

What follows is documentation of each verification test performed for the digital Theremin example.

Object: touch
Test Number: 1

Test Description:

Add debug code to show the X,Y readings of a touch. Touch all four corners of the screen. Write down the X,Y coordinates of each corner of the screen.

Test Outcome:

(254,3772) (3882,3704)

(178,233) (3851,158)

One could hope for better, but this is what you get with a cheap resistive touch screen.

Object: touch
Test Number: 2

Test Description:

Verify that the max amplitude returned by touch_amplitude() is within 16 counts of 255, and that the minimum amplitude returned by touch_amplitude() is less than 16.

Test Outcome:

Max amplitude: 240

Min amplitude: 92

The minimum amplitude is large because the minimum X value is large. It might be possible to use the screen corner values to provide more accurate scaling of the amplitude, but that will not be done in this demonstration

program.

Object: touch
Test Number: 3

Test Description:

Verify that that the max frequency returned ty touch_frequency() is within 20 Hz of 1047.

Test Outcome:

The maximum frequency computed from the Y touch value is 967, certainly not the high C value we were looking for. But once again calibration based upon the touchscreen corner values could be used to make the highest frequency more accurate.

Object: touch
Test Number: 4

Test Description:

Verify that the minimum frequency returned by touch_frequency() is within 10 Hz of 44 H

Test Outcome:

The minimum frequency returned by the touch screen reading was 80Hz. Not too bad, but certainly not good enough for musical accuracy without further calibration efforts.

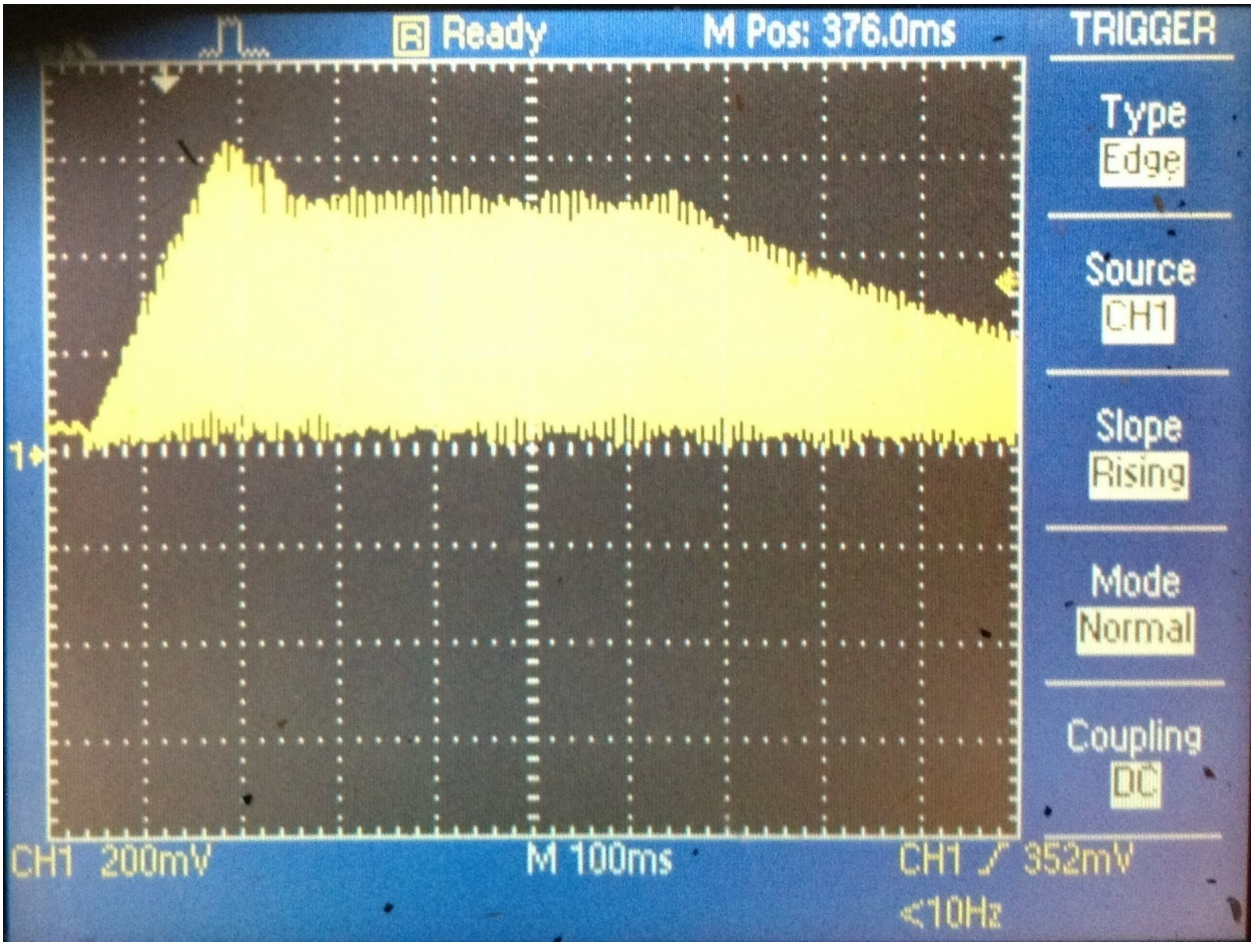
Object: note

Test Number: 1

Test Description:

Hook up an oscilloscope to the audio output jack. Verify that notes which are not interrupted before the end of the release phase go through the four phases shown in the diagram.

Test Outcome:

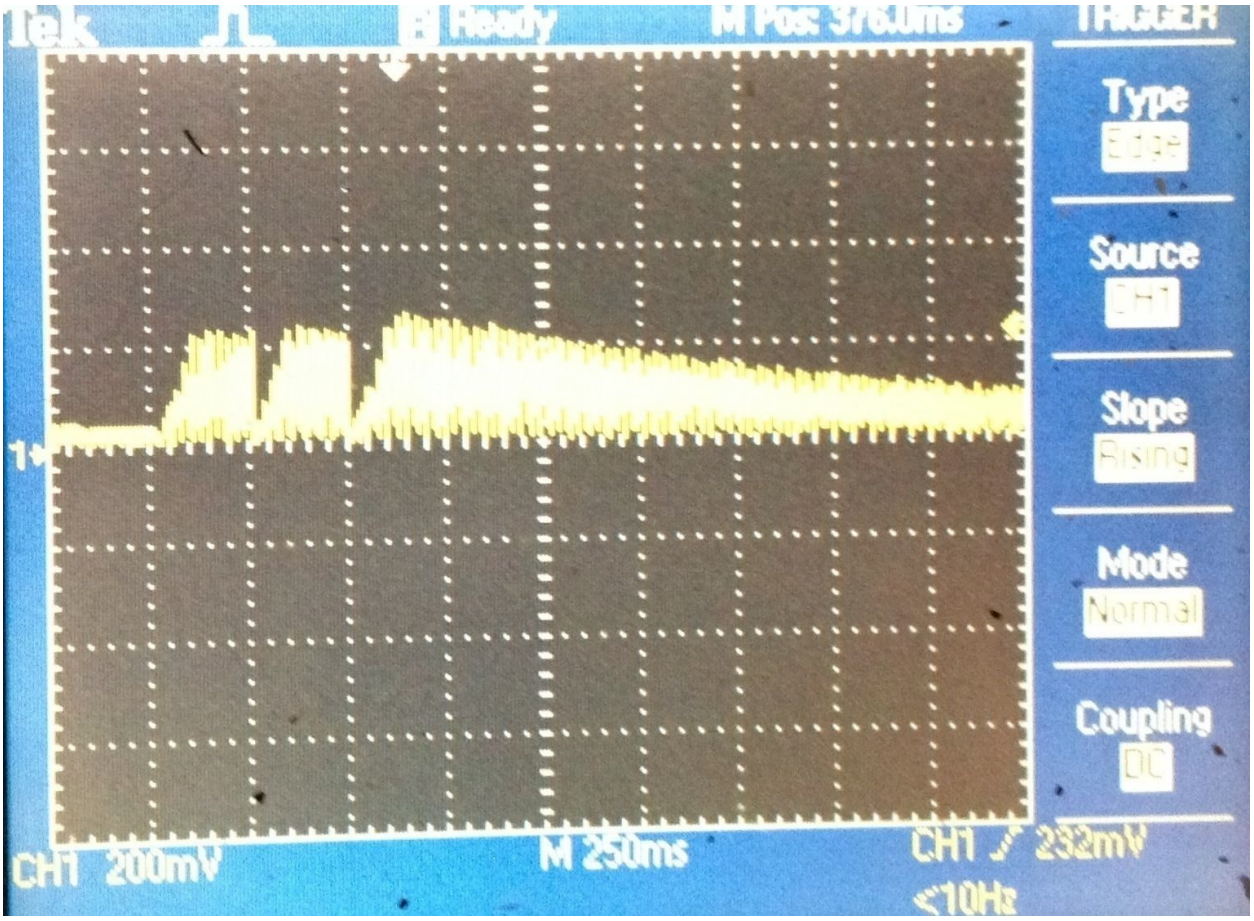


Object: note
Test Number: 2

Test Description:

Verify that notes which succeed each other in quick succession, without a complete release phase, start by cutting off the preceeding note, and then beginning the attack phase.

Test Outcome:



Object: note
Test Number: 3

Test Description:

Verify that the note amplitude increases as the X value of the touch screen increases, and that the frequency of the note increases as the Y value from the touch screen increases.

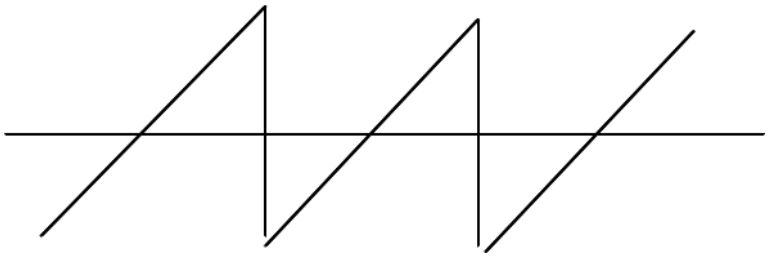
Test Outcome:

Yes, that is what happens so long as the touch screen connector is on the lower left hand side of the touch screen.The X coordinate then grows to the right, and the Y coordinate grows upward.

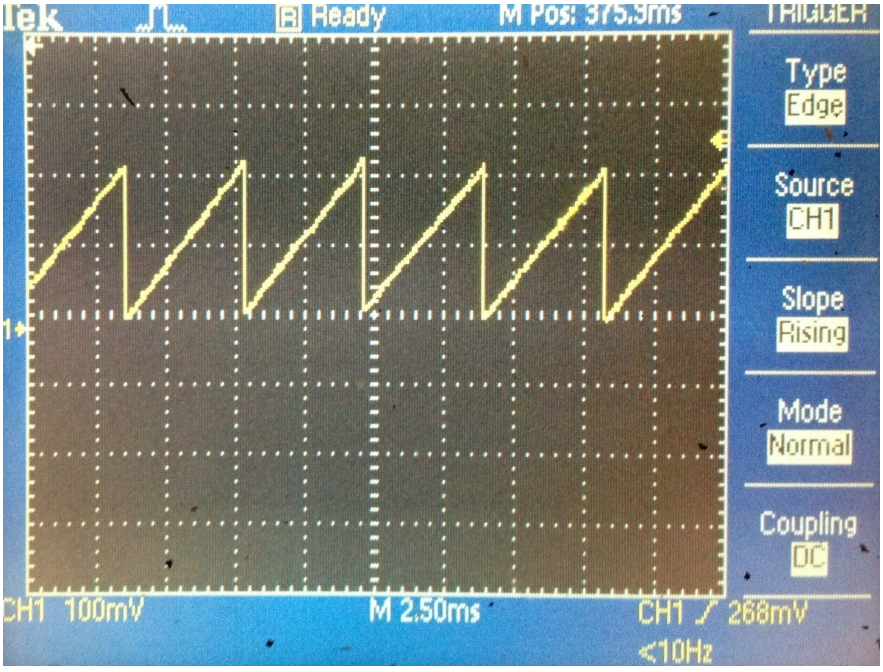
Object: wave
Test Number: 1

Test Description:

Hook an oscilloscope up to the audio output jack. Verify that immediately after powerup the output is a sawtooth waveform, as shown below



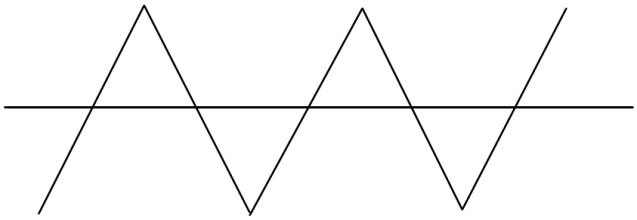
Test Outcome:



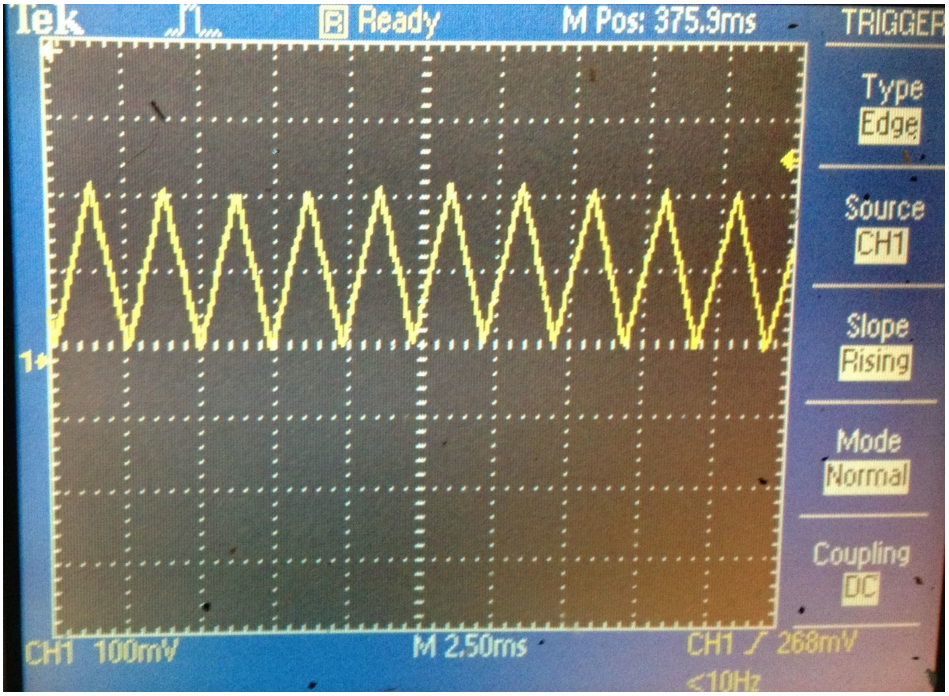
Object: wave
Test Number: 2

Test Description:

Verify with the oscilloscope that the sawtooth waveform changes to a triangle waveform the first time the white button on the application button is pressed inward in its center position.



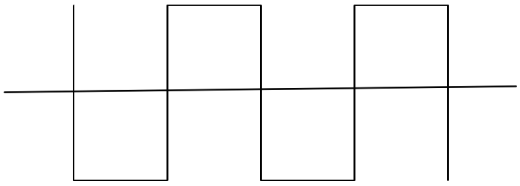
Test Outcome:



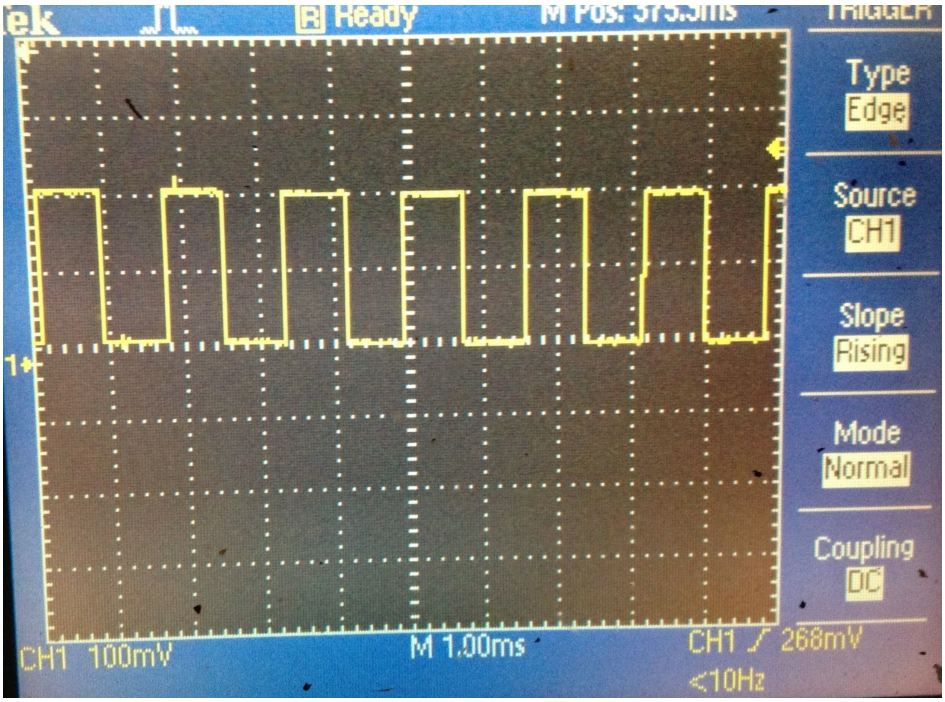
Object: wave
Test Number: 3

Test Description:

Verify with the oscilloscope that the triangle waveform changes to a square waveform the next time the white button on the application button is pressed inward in its center position.



Test Outcome:

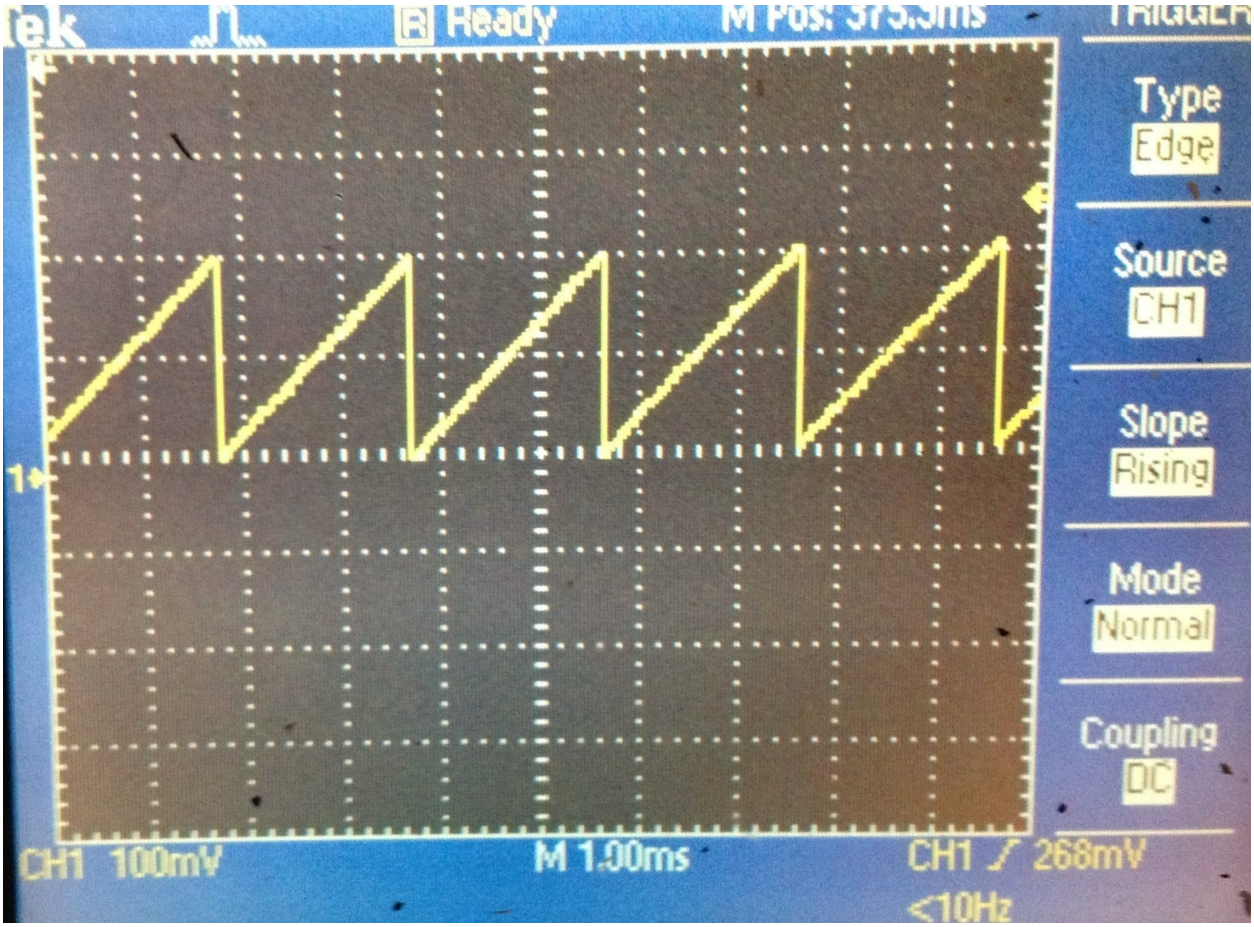


Object: wave
Test Number: 4

Test Description:

Verify with the oscilloscope that the square waveform changes back to a sawtooth waveform the next time the white button on the application button is pressed inward in its center

Test Outcome:

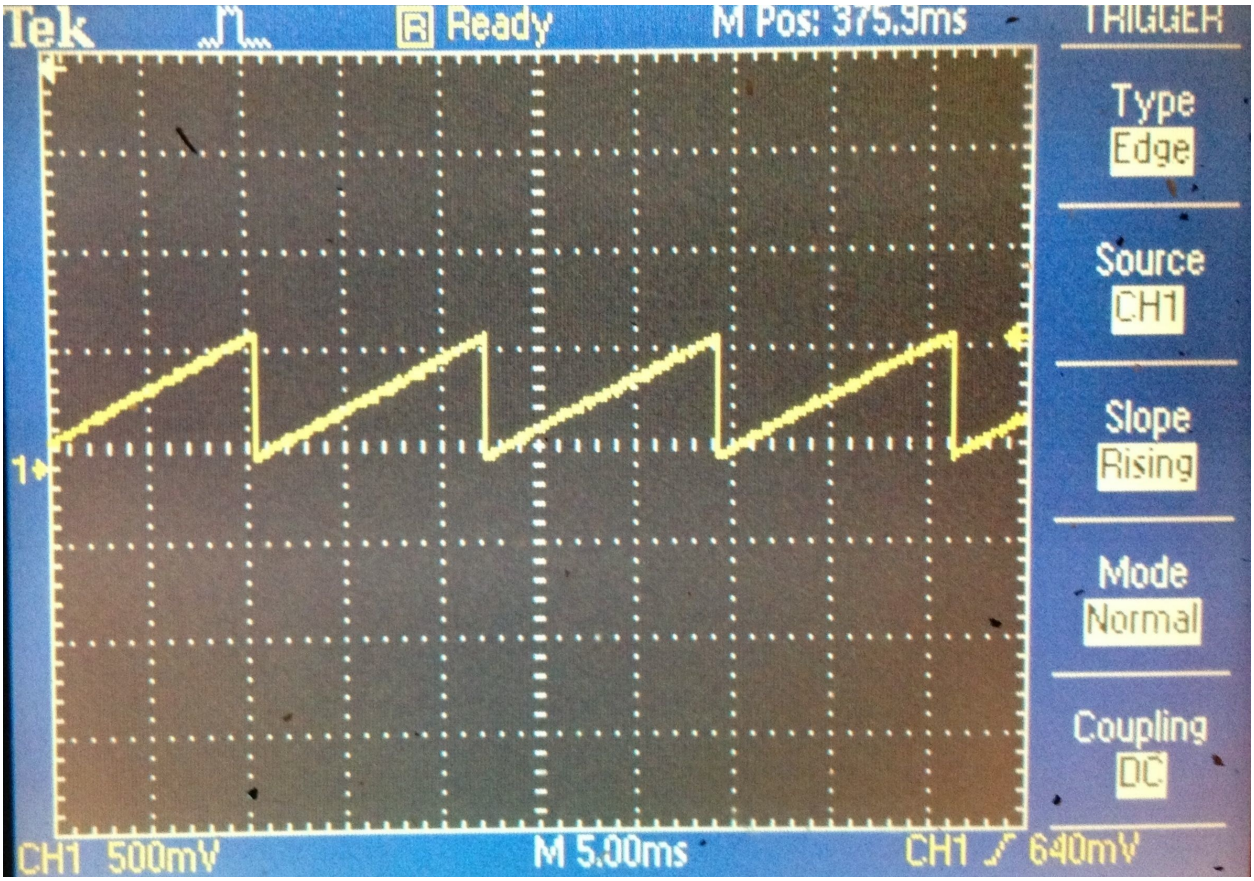


Object: wave
Test Number: 5

Test Description:

Verify with the oscilloscope that the minimum frequency that can be output using the touchscreen is within 10Hz of 43 Hz.

Test Outcome:



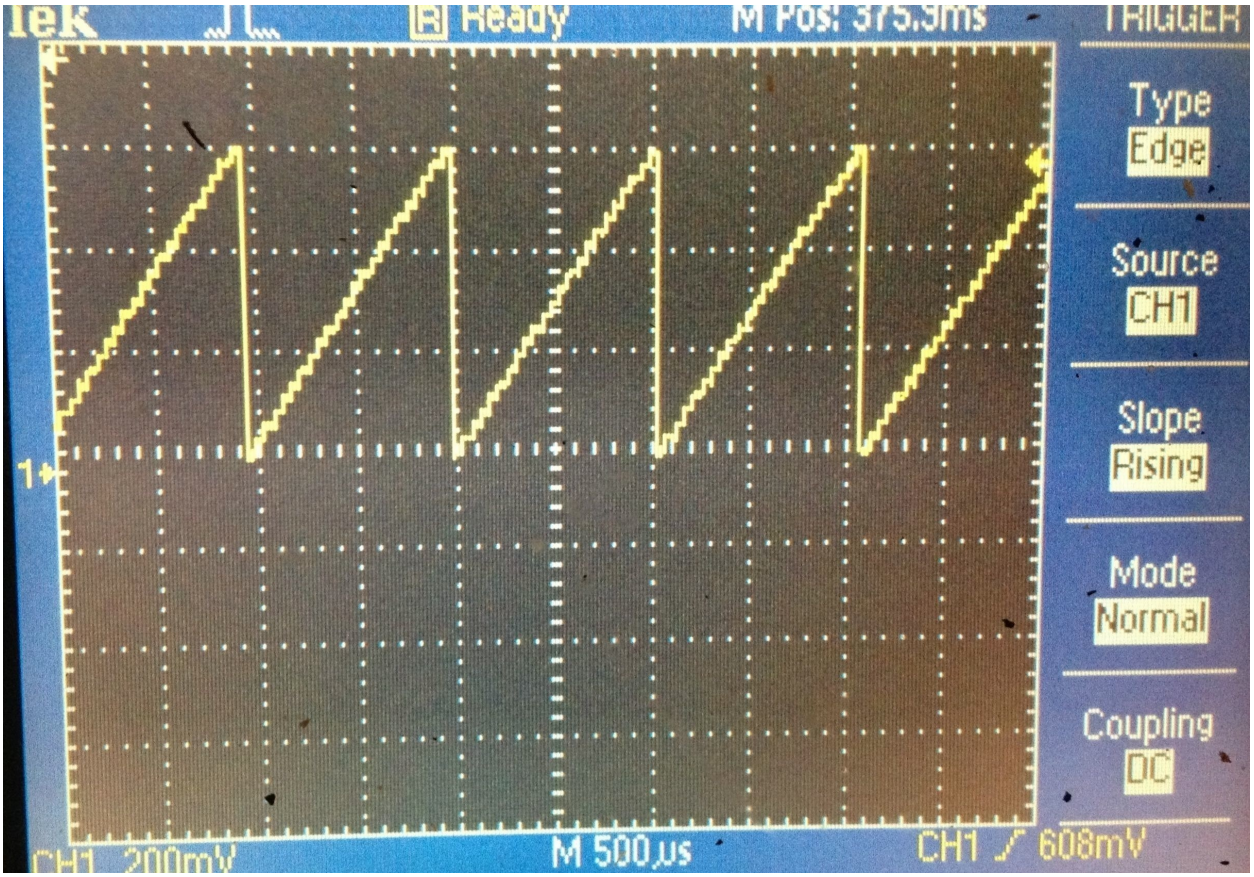
As low a frequency as we could get.

Object: wave
Test Number: 6

Test Description:

Verify with the oscilloscope that the maximum frequency that can be output using the touchscreen is within 20 Hz of 1047 Hz.

Test Outcome:



As high a frequency as we could get

Object: envlp
Test Number: 1

Test Description:

Hook an oscilloscope up to the audio output jack. Verify that the output waveform does not develop discontinuities (other those expected one at pi or 2*pi radians) as the stylus is moved around on the touch screen varying envelope amplitude and frequency.

Test Outcome:

Continuously varying amplitude and frequency with the touchscreen showed no unexpected discontinuities in the waveform or sustain phase envelop displayed on the oscilloscope. Neither were cracks or pops heard while the amplitude and frequency were varied.

Object: jswitch
Test Number: 1

Test Description:

Use debug facility to print the value of official every time it changes. Press some joystick buttons and verify the behavior of the variable called official. Make sure that the bits set for each switch are:

```
#define JS_CENTER 0x10
#define JS_UP     0x04
#define JS_DOWN   0x08
#define JS_LEFT   0x01
#define JS_RIGHT  0x02
```

Test Outcome:

Center: 0x10 Up: 0x04 Down: 0x08 Left: 0x01 Right: 0x02

Object: dma
Test Number: 1

Test Description:

The buffer output rate from the dma is 43/sec.

Plug earphones into the output jack of the application board. Listen to the output at various note frequencies. Verify that there is no significant 43 Hz flutter in the audio output, as might be heard if the dma buffer switching were not operating properly.

Test Outcome:

No significant 43Hz flutter was heard in the output audio.

VALIDATE

Verification is a double check that the system's virtual objects perform and interact as they are designed to. Validation is a series of checks to verify that the system as a whole meets its requirements.

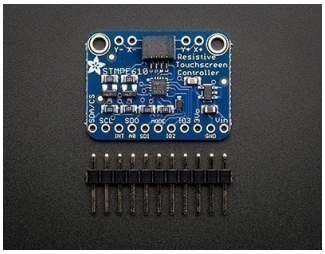
The validation report for the Digital Theremin system evaluates each requirement in the formal requirements list against the performance of the system. The formal requirements are listed below in italics, with discussion and the results of testing below each requirement.

1) Requirement: The hardware for the Theremin project has already been chosen, primarily for its ready availablity, and superior compiler and library support. It includes an mbed.org Microcontroller board with an LPC1768 ARM Cortex-M3 Processor.

That 40pin DIP board will be plugged into an mbed Application Board, which provides additional peripherals, such as a five-position joy-switch, and pull-up resistors for a I2C lines SDA and SCL, which will connect to a

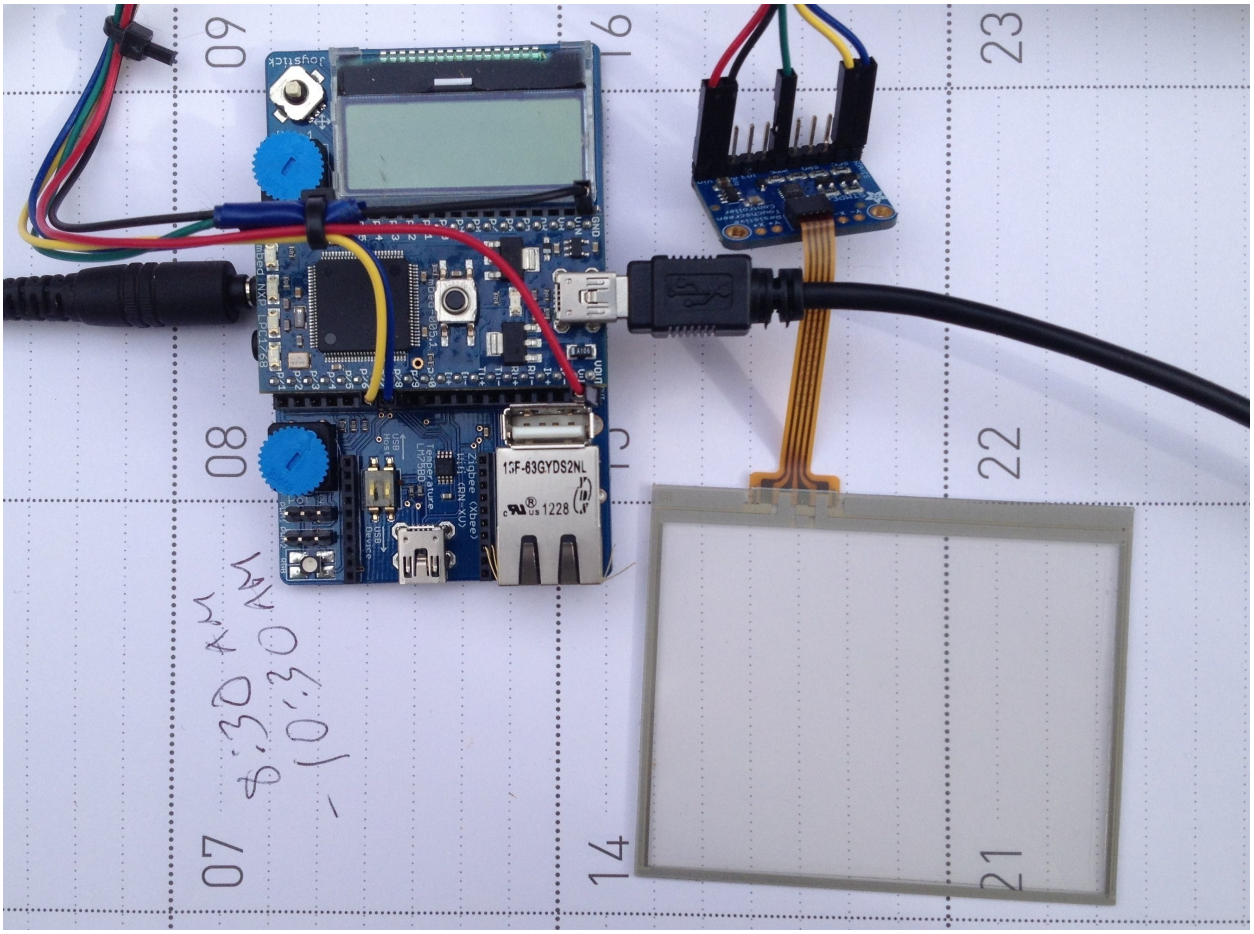
touchscreen controller chip.

Both the processor board and application board may be purchased from www.sparkfun.com or www.adafruit.com. The touchscreen controller board uses the SMTPE610 controller chip, and is available from adafruit.com.



Any 4-wire resistive touchscreen can be used, but the one purchased for this example is roughly 2.2 by 2.75 inches, and has a connector compatible with the touchscreen controller board. It is similar to the touchscreen, from adafruit.com, shown below:

1) Validation: Below is a photo of the completed hardware configuration:



2) Requirement: When powered up, the Therimen briefly enters an initialization period, during which it sets up the system hardware and computes waveform table(s). Then it enters normal operation using a saw waveform as it's default.

2) Validation: If the program produces the expected waveforms after power-up, we can assume it was properly initialized and computed at least the first waveform table. The waveform should then change each time the joy-switch is center-pressed.

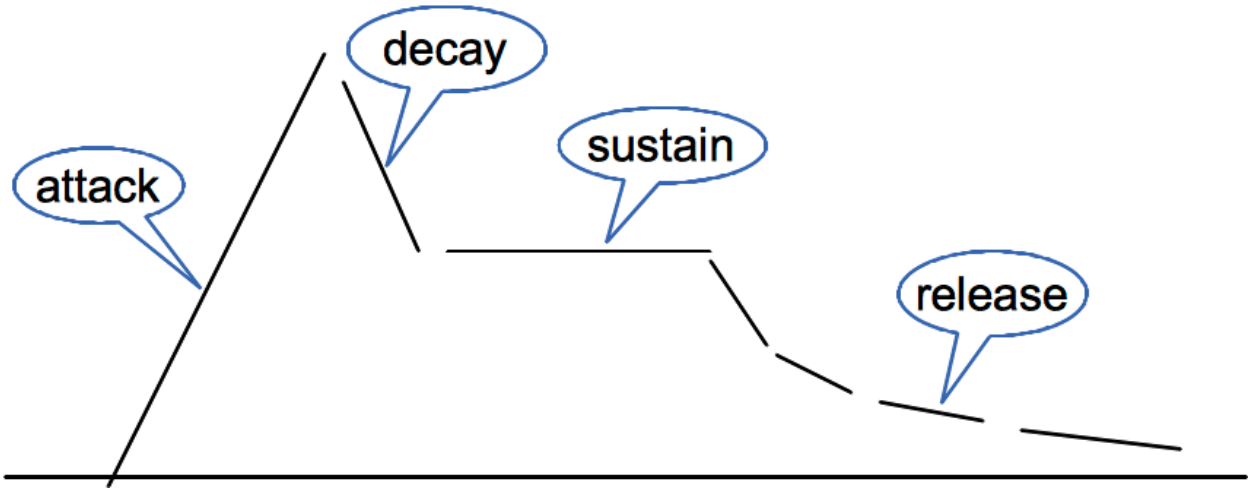
This was demonstrated in the Verification of the wave object.

3) Requirement: Every note played consists of a waveform modulated by an envelope. The waveform has the frequency and volume determined by the Y-X touch position. The envelope modulation evolves in time through four separate stages: attack, decay, sustain, and release. The release stage begins when the touch is removed.

3) Validation: Touch the touch screen with a finger or a stylus and make sure that the frequency increases when the finger is moved in the positive Y direction and decreases when moved in the negative Y direction. Also make sure the volume increases when the finger is moved in the positive X direction, and decreases when the finger is moved in the negative X direction.

That was done and the volume and frequency change as specified.

Next hook up an oscilloscope to the audio output and make sure the envelop progresses through the four stages shown below:



This was demonstrated in Test Number 1 of note verification.

4) Requirement: In normal operation the Theremin accepts input from the resistive touchscreen and the joy-switch. It reacts to a touch by triggering a

note of a frequency and volume determined by the Y & X position of the touch. It reacts to a center press of the joy-switch by making the next waveform available for the next note or notes. It reacts to left, right, up, and down movements of the joy-switch to shorten release phase, lengthen release phase, lengthen attack phase, and shorten attack phase of the next note or notes.

4) Validation: *Touch the screen and demonstrate that the Theremin reacts by producing a tone that varies in pitch and amplitude with the Y and X position of the touch: Proved in Validation item 3.*

Push the joy-switch up repeatedly and verify that it softens the attack sound.

Yes it does.

Push the joy_switch down repeatedly and verify that it hardens the attack sound.

Yes it does.

Push the joy switch left and verify that it shortens the release duration.

Yes it does.

Push the joy switch right and verify that it lengthens the release duration.

Indeed it does.

Press the joy-switch straight in and verify that the waveform changes.

Yes it does.

APPENDIX A -- CELLULAR CALCULATOR SCRIPTS FOR ESTIMATING

For a windows XP program which interprets these scripts, as well as the executable form of the scripts, email the author at david@exopiped.com.

SCRIPT FOR COCOMOII COMPUTATION

Cellname: ACAP
Cellvalue: 0.86
Cell Comments:
Analyst Capability

Rate according to scale below. Place rating into value field of this cell.

Extra Low ----
Very Low 1.46
Low 1.19
Nominal 1.00
High 0.86
Very High 0.71
Extra High ----
Cell Script:

Endcell: ACAP

Cellname: AEXP
Cellvalue: 0.91
Cell Comments:
Application Experience

Rate according to scale below. Place rating into value field of this cell.

Extra Low ----
Very Low 1.29
Low 1.13
Nominal 1.00
High 0.91
Very High 0.82

Extra High ----
Cell Script:

Endcell: AEXP

Cellname: CPLX
Cellvalue: 1.00
Cell Comments:
Product Complexity

Rate according to scale below. Place rating into value field of this cell.

Extra Low ----
Very Low 0.70
Low 0.85
Nominal 1.00
High 1.15
Very High 1.30
Extra High 1.65
Cell Script:

Endcell: CPLX

Cellname: DATA
Cellvalue: 0.94
Cell Comments:
Database Size.

Rate according to scale below. Place rating into value field of this cell.

Extra Low ----
Very Low ----
Low 0.94
Nominal 1.00
High 1.08
Very High 1.16
Extra High ----
Cell Script:

Endcell: DATA

Cellname: Embedded
Cellvalue:
Cell Comments:
The embedded mode of software development describes projects where there is only a general understanding of product objectives, moderate experience in working with related systems, a full need for conformance with pre-established requirements and external interface specs, full concurrent development of hardware and procedures, considerable need for innovative data processing architectures and algorithms, and a high premium on early completion. This mode can be for all product size ranges.
Cell Script:

Endcell: Embedded

Cellname: Enhanced
Cellvalue:
Cell Comments:
To enhance your estimate, fill in values for the following factors, using the guidance from the text field of each factor's cell. Then execute the script for this cell.
RELY DATA CPLX RQUT TIME STOR VIRT TURN ACAP AEXP PCAP VEXP LEXP MODP TOOL SCED
Cell Script:
```
out("\nEnhanced Estimate")
set MM to EnhancedMM!*TotEffect!
out("\nMM=",MM," man months, TDEV=",TDEV!," calendar months, FSP=",FSP!," developers required")
```

Endcell: Enhanced

Cellname: EnhancedMM
Cellvalue: 2.53
Cell Comments:

Cell Script:

```
if(mode=1) {
   3.2*pow(KDSI,1.05)
} elseif(mode=2) {
   3.0*pow(KDSI,1.12)
} elseif(mode=3){
   2.8*pow(KDSI,1.20)
} else {
   out("\nPlease choose a mode of 1,2, or 3")
   exit
}
```

Endcell: EnhancedMM

Cellname: FSP
Cellvalue: 0.60
Cell Comments:
This is the number of equivalent Full Time Software Developers needed to achieve the development time computed in TDEV.
Cell Script:
MM/TDEV
Endcell: FSP

Cellname: home
Cellvalue:
Cell Comments:
The Cocomo model estimates the total effort, schedule, and number of developers required to develop a software product, given the estimated number of thousands of lines of delivered source instructions (KDSI) and the mode of software development.

The estimates provided are:
MM -- the total effort required in man months
TDEV -- schedule time required assuming optimum staffing
FSP -- Full time software developers needed for optimum staffing.

You must decide whether the project takes place in Organic, Semidetached, or Embedded mode and goodness knows. Visit the three cells and set mode's value to 1 for Organic,

2 for Semidetached, or 3 for Embedded.

Enter the number of thousands of estimated delivered source lines, and store the value in KDSI.

Then return here and select Run from the Actions menu.

Go to cell Enhanced for instructions on generating enhanced estimates based upon a number of detailed factors.

Cell Script:
out("\nMM=",MM!," man months, TDEV=",TDEV!," calendar months, FSP=",FSP!," developers required")

Endcell: home

Cellname: KDSI
Cellvalue: .8
Cell Comments:
Source statements are all program instructions created by project personnel and processed into machine code by compilers, etc. Delivered source statements excludes nondelivered support software such as test drivers. In the value field of this cell place the number of thousands of delivered source statements expected.
Cell Script:

Endcell: KDSI

Cellname: LEXP
Cellvalue: 0.95
Cell Comments:
Programming Language Experience

Rate according to scale below. Place rating into value field of this cell.

Extra Low ----
Very Low 1.14
Low 1.07

Nominal 1.00
High 0.95
Very High ----
Extra High ----
Cell Script:

Endcell: LEXP

Cellname: MM
Cellvalue: 1.90
Cell Comments:
This cell computes the expected number of man months(152 hrs)
required to produce the number of thousands of delivered
source statements given in KDSI.
Cell Script:

```
if(mode=1) {
   2.4*pow(KDSI,1.05)
} elseif(mode=2) {
   3.0*pow(KDSI,1.12)
} elseif(mode=3){
   3.6*pow(KDSI,1.20)
} else {
   out("\nPlease choose a mode of 1,2, or 3")
   exit
}
```

Endcell: MM

Cellname: mode
Cellvalue: 1
Cell Comments:
The mode of the software development project will be either
&Organic, &Semidetached, or &Embedded. Visit the cells by
those names and decide which mode applies for your project.
Then set the value of this cell to:
   1 for Organic
   2 for Semidetached
or
   3 for Embedded
Cell Script:

Endcell: mode

Cellname: MODP
Cellvalue: 0.91
Cell Comments:
Modern Programming Practices

Rate according to scale below. Place rating into value field of this cell.

Extra Low ----
Very Low 1.24
Low 1.10
Nominal 1.00
High 0.91
Very High 0.82
Extra High ----
Cell Script:

Endcell: MODP

Cellname: Organic
Cellvalue:
Cell Comments:
The organic mode of software development describes projects where there is a thorough understanding of product objectives, there is extensive experience in working with related software systems, there is a basic need for conformance with published requirements and interface specs, some concurrent development of new hardware and operational procedures, minimal need for innovative architectures and algorithms, a low premium on early completion, and less than 50 KDSI.

Cell Script:

Endcell: Organic

Cellname: Output
Cellvalue:

Cell Comments:

ACAP
AEXP
CPLX
DATA
Embedded
Enhanced
EnhancedMM
FSP
home
KDSI
LEXP
MM
mode
MODP
Organic
PCAP
RELY
RQUT
SCED
Semidetached
STOR
TDEV
TIME
TOOL
TotEffect
TURN
VEXP
VIRT
Cell Script:

Endcell: Output

Cellname: PCAP
Cellvalue: 0.86
Cell Comments:
Programmer Capability

Rate according to scale below. Place rating into value field

of this cell.

Extra Low ----
Very Low 1.42
Low 1.19
Nominal 1.00
High 0.86
Very High 0.70
Extra High ----
Cell Script:

Endcell: PCAP

Cellname: RELY
Cellvalue: 1.00
Cell Comments:
Required reliability.

Rate according to scale below. Place rating into value field of this cell.

Extra Low ----
Very Low 0.75
Low 0.88
Nominal 1.00
High
Very High 1.15
Extra High 1.40
Cell Script:

Endcell: RELY

Cellname: RQUT
Cellvalue: 1.0
Cell Comments:
Requirement Changes

Rate according to scale below. Place rating into value field of this cell.

Extra Low ----
Very Low ----
Low 0.98
Nominal 1.00
High 1.50
Very High 2.00
Extra High 2.50
Cell Script:

Endcell: RQUT

Cellname: SCED
Cellvalue: 1.08
Cell Comments:
Schedule Pressures

Rate according to scale below. Place rating into value field of this cell.

Extra Low ----
Very Low 1.23
Low 1.08
Nominal 1.00
High 1.04
Very High 1.10
Extra High ----
Cell Script:

Endcell: SCED

Cellname: Semidetached
Cellvalue:
Cell Comments:
The semidetached mode of software development describes a project in which there is considerable understanding of product objectives, considerable experience in working with related software systems, considerable need for conformance with pre-established requirements and external interface specifications, considerable concurrent development of hardware and procedures, some need for innovative data

processing architectures and algorithms, a medium premium on early completion, and between 50 and 300 KDSI.
Cell Script:

Endcell: Semidetached

Cellname: STOR
Cellvalue: 0.98

Cell Comments:
Main Storage Constraint

Rate according to scale below. Place rating into value field of this cell.

Extra Low ----
Very Low ----
Low 0.98
Nominal 1.00
High 1.06
Very High 1.21
Extra High 1.56
Cell Script:

Endcell: STOR

Cellname: TDEV
Cellvalue: 3.19
Cell Comments:
This cell compute the number of months estimated for software development, given he number of man months of effort required. This assumes the optimum staffing, which can be computed from the cell FSP.
Cell Script:

```
if(mode=1) {
    2.5*pow(MM,0.38)
} elseif(mode=2) {
   2.5*pow(MM,0.35)
} elseif(mode=3) {
   2.5*pow(MM,0.32)
```

```
} else {
    out("\nInvalid Mode")
    exit
}
Endcell: TDEV

Cellname: template
Cellvalue:
Cell Comments:
Extra Low
Very Low
Low
Nominal
High
Very High
Extra High
Cell Script:

Endcell: template

Cellname: TIME
Cellvalue: 1.11
Cell Comments:
Execution Time Constraint

Rate according to scale below.  Place rating into value field
of this cell.


Extra Low  ----
Very Low   ----
Low        0.98
Nominal    1.00
High       1.11
Very High  1.30
Extra High 1.65
Cell Script:

Endcell: TIME
```

Cellname: TOOL
Cellvalue: 0.91
Cell Comments:
Use of Software Tools

Rate according to scale below. Place rating into value field
of this cell.

Extra Low ----
Very Low 1.24
Low 1.10
Nominal 1.00
High 0.91
Very High 0.83
Extra High ----
Cell Script:

Endcell: TOOL

Cellname: TotEffect
Cellvalue: 0.52
Cell Comments:

Cell Script:
RELY * DATA * CPLX * RQUT * TIME * STOR * VIRT * TURN * ACAP
* AEXP * PCAP * VEXP * LEXP * MODP * TOOL * SCED
Endcell: TotEffect

Cellname: TURN
Cellvalue: 0.87
Cell Comments:
Computer Turnaround Time

Rate according to scale below. Place rating into value field
of this cell.

Extra Low ----
Very Low ----
Low 0.87
Nominal 1.00

High 1.07
Very High 1.15
Extra High ----
Cell Script:

Endcell: TURN

Cellname: VEXP
Cellvalue: 1.0
Cell Comments:
Virtual machine Experience

Rate according to scale below. Place rating into value field of this cell.

Extra Low ----
Very Low 1.21
Low 1.10
Nominal 1.00
High 0.90
Very High ----
Extra High ----
Cell Script:

Endcell: VEXP

Cellname: VIRT
Cellvalue: 1.0
Cell Comments:
Virtual Machine Volatility

Rate according to scale below. Place rating into value field of this cell.

Extra Low ----
Very Low ----
Low 0.87
Nominal 1.00
High 1.15
Very High 1.30

Extra High ----
Cell Script:

Endcell: VIRT

SCRIPT FOR FUNCTION POINT COMPUTATION

Script written by Kevin J. Northover
Refs: Capers Jones, IEEE Computer, March 1996, 116-118.
Ted Lewis, IEEE Computer, August 1996, 13-15.

Cellname: Bugfix
Cellvalue: 0.30
Cell Comments:
Rule 6: Each software review, inspection, or test step will find and remove 30 percent of the bugs that are present.
Cell Script:

Endcell: Bugfix

Cellname: BugsLeft
Cellvalue: 4.52132
Cell Comments:
Uses Rule 6 &Bugfix to estimate the number of defects that will be shipped assuming &TestRun test cycles.
Cell Script:
Defect!*pow(1-Bugfix,TestRun)
Endcell: BugsLeft

Cellname: Creep
Cellvalue: 0.01
Cell Comments:
Rule 3: Creeping user requirements will grow at an average rate of 1 percent per month over the entire development schedule.

Enter assumed monthly creep rate in the value field.
Cell Script:

Endcell: Creep

Cellname: Critical_Creep
Cellvalue: 0.14637
Cell Comments:
This cell computes the monthly creep rate (fractional change in requirements) for which the growth in requirements over the build time for the project equals the size of the original specification.
Cell Script:
exp(ln(2)/pow(fp, 0.4)) - 1
Endcell: Critical_Creep

Cellname: Defect
Cellvalue: 160.06080
Cell Comments:
Rule 5: Raising the number of function points to the 1.25 (1.27) power predicts the approximate defect potential for new (enhancement) software projects.
Cell Script:
pow(fp,1.25)

Endcell: Defect

Cellname: Effort
Cellvalue: 1.96204
Cell Comments:
Rule 10: Multiply software development schedules by number of personnel to predict the approximate number of months of staff effort.

Commonly define a staff month as 22 working days with six productive hours each day, or 132 work hours per month
Cell Script:
Schedule!*Staffing!
Endcell: Effort

Cellname: fp
Cellvalue: 58

Cell Comments:
Number of Function Points in the project.

Cell Script:
```
if (KDSI > 0) {
  KDSI*1000/LOC_to_fpoint
}
```
Endcell: fp

Cellname: Growth
Cellvalue: 5.17868
Cell Comments:
Percentage Growth in Requirements by delivery for &Creep rate
Cell Script:
```
100*(pow(1+Creep,Schedule!)-1)
```
Endcell: Growth

Cellname: home
Cellvalue:
Cell Comments:
Software Estimating Rules of Thumb.
This web generates a number of estimates for sizing software projects based on the function point model. These are very rough estimates to be used with CAUTION.

To use either: a) enter the number of thousands of lines of delivered source code &KDSI; or b) set KDSI to 0 and enter the number of function points &FP. Then return to this home cell and execute.

For more refined estimates consider varying the conversion from source lines to function points &LOC_to_fpoint. Or explore the creep rate parameter &Creep.

To investigate the number of Quality Control steps required vary the &TestRun parameter.

Script written by Kevin J. Northover
Refs: Capers Jones, IEEE Computer, March 1996, 116-118.

Ted Lewis, IEEE Computer, August 1996, 13-15.

Cell Script:
```
out("1. Function Points       = ",fp!,"\n")
out("2. Paper Deliverables  = ",paper!," pages, ",400*paper!,
" words\n")
out("3. Fnct. Pts. Growth    = ",Growth!,
" percent by delivery \n")
out("3a.  Critical Creep       = ",Critical_Creep!*100,
" percent\n")
out("4. Test Cases              = ",test!,";  test runs = ",
test!*4,"\n")
out("5. Defect Potential      = ",Defect!,"\n")
out("6. Defects Shipped       = ",BugsLeft!,"\n")
out("7. Schedule (months)  = ",Schedule!,"\n")
out("8. Build Staffing           = ",Staffing!,"\n")
out("9. Maintenance Staff  = ",Maintenance!,"\n")
out("9b. Years of Use          = ",Lifetime!,"\n")
out("10. Total Effort           = ", Effort!,"\n")
```
Endcell: home

Cellname: KDSI
Cellvalue: 0
Cell Comments:
Enter the estimated number of thousands of lines of delivered noncommentary logical source code statements in the value field of this cell.
Cell Script:

Endcell: KDSI

Cellname: Lifetime
Cellvalue: 2.75967
Cell Comments:
Rule 9b: Raising the function points total to the 0.25 power yields the approximate number of years the application will stay in use.
Cell Script:
```
pow(fp,0.25)
```
Endcell: Lifetime

Cellname: LOC_to_fpoint
Cellvalue: 100
Cell Comments:
Rule 1: One function point = 100 logical source code statements.
This ratio can vary from >300 for assembler to <20 for object oriented languages and program generators. For procedural languages such as COBOL, C, Fortran, etc. 100 is a rough conversion factor.
Enter the conversion factor to use in the value field.
Cell Script:

Endcell: LOC_to_fpoint

Cellname: Maintenance
Cellvalue: 0.11600
Cell Comments:
Rule 9: Dividing the number of function points by 500 predicts the approximate number of maintenance personnel required to keep the application updated.
Cell Script:
fp/500
Endcell: Maintenance

Cellname: Paper
Cellvalue: 106.64542
Cell Comments:
Rule 2: Raising the number of function points to the 1.15 power predicts approximate page counts for paper documents associated with software projects.
Cell Script:
pow(fp,1.15)
Endcell: Paper

Cellname: Schedule
Cellvalue: 5.07424
Cell Comments:
Rule 7: Raising the number of function points to the 0.4 power predicts the approximate development schedule in

calendar months.
Cell Script:
pow(fp,0.4)
Endcell: Schedule

Cellname: Staffing
Cellvalue: 0.38667
Cell Comments:
Rule 8: Dividing the number of function points by 150 predicts the approximate number of personnel required for the application.
Cell Script:
fp/150
Endcell: Staffing

Cellname: Test
Cellvalue: 130.65126
Cell Comments:
Rule 4: Raising the number of function points to the 1.2 power predicts the approximate number of test cases created.
Cell Script:
pow(fp,1.2)
Endcell: Test

Cellname: TestRun
Cellvalue: 10
Cell Comments:
Number of Test Cycles in development of system. Includes Major Design Reviews, code inspections and various testing levels.
Cell Script:

Endcell: TestRun

APPENDIX B -- CODE FOR VIRTUAL OBJECTS

```
/*
 * main.cpp
 * Sending digital note waveforms out the onboard DAC
 * in responses to touches on an XY pad(resistive touchscreen)
 * X changes audio volume..  Y changes audio frequency
 * Varies from forty Hz to 1047 Hz (C6)
 */
#include "mbed.h"
#include "debug.h"
#include "dma.h"
#include "wave.h"
#include "note.h"
#include "jswitch.h"
#include "touch.h"
#include "envlp.h"

int main(void)
{
    js_init();  // initialize joystick
    envlp_init(); // initialize envelope parameters
    wave_init();  // starts with default wave type
    touch_init();  // initialize touchscreen controller
    note_init();    // initialize note production

    while (1==1) {
        js_debounce();  // debounce joystick
        envlp_update(); // update envelope parameters
        wave_update();  // update waveform choice
        note_update();  // update note or start a new one
        wait_ms(2);  // lower limit on loop timing
    }
}
```

```
/*
 * jswitch.h -- header for jswitch.cpp, which reads the
 *    joystick pins, and provides debounced version
 *    of the pins to js_read() caller.
 */
#ifndef JSTICK_H
#define JSTICK_H
#define JS_CENTER 0x10
#define JS_UP     0x04
#define JS_DOWN   0x08
#define JS_LEFT   0x01
#define JS_RIGHT  0x02
void js_init(void);
void js_debounce(void);
unsigned char js_read(unsigned char mask);
#endif
```

```
/*
 * jswitch.cpp -- debounces and reads pins 12
 *    (SW_DOWN), 13(SW_LEFT),14(SW_CENTER),15(SW_UP),
 *    and 16(SE_RIGHT) of the mbed 40 pin dip board
 */
#include "mbed.h"
#include "debug.h"
#include "jswitch.h"

#define JS_BLANKING_MSEC 333

DigitalIn js_down(p12);
DigitalIn js_left(p13);
DigitalIn js_center(p14);
DigitalIn js_up(p15);
DigitalIn js_right(p16);

Timer js_timer;

static unsigned char js_oldest=0;
static unsigned char js_older=0;
static unsigned char js_old=0;
static unsigned char js_now=0;
static unsigned char js_official=0;
static int js_msec=0;

void js_init(void)
{
    js_official = 0;
```

```
    js_now = 0;
    js_old = 0;
    js_older = 0;
    js_oldest = 0;
    js_timer.start();
}
/*
 * js_debounce -- button presses will not be reported
 * any more that once every 333 msec.
 */
void js_debounce(void){
    if ((js_timer.read_ms()-js_msec)>JS_BLANKING_MSEC) {
        if (js_official==0) {
            js_oldest = js_older;
            js_older=js_old;
            js_old=js_now;
            js_now = ( (js_center << 4)
                     | (js_up << 3)
                     | (js_down << 2)
                     | (js_left << 1)
                     | js_right );
            js_official|=(js_now&js_old&js_older&js_oldest);
            if (js_official>0) {
                js_msec=js_timer.read_ms();
            }
        } else {
            js_now=0;
            js_old=0;
            js_older=0;
            js_oldest=0;
        }
    }
}
/*
 *   js_read -- reads buttons indicated by the mask.
 *   once read the masked button indications are cleared
 */
unsigned char js_read(unsigned char mask)
{
    unsigned char retval;

    retval = js_official & mask;
    js_official &= ~retval;
    return retval;
}
```

```
/*
 * touch.cpp -- touch screen monitor
 */
#include "mbed.h"
#include "debug.h"
#include "touch.h"

#define NIL (-1)
/*
 * Connecting the touch screen
 *    STMPE610 MODE pin is tied low for I2C interface
 *    STMPE610 A0 pin tied low for I2C address 0x41
 *                        (0x82 when shifted left 1)
 *    STMPE610 A0 pin tied high for I2C address 0x44
 *                         (0x88 when shifted left 1)
 *
 *    I2C works on board pins 9 and 10. (SDA, SDL)
 *     ( STMPE610 pins SDAT and  SCLK )
 *
 * Setting up the touch screen
 *
 * Disable Touch Screen and A/D clock -- SYS_CTRL2
 *    Set TSC_OFF bit high, ADC_OFF bit high
 *
 * ConfigureTouch Screen -- TSCFG register
 *    Set up for 4 sample averaging
 *    Touch detect delay of 1 msec
 *    Settling time of 1 msec
 *
 * Touchscreen Window -- WDW_TR_X, WDW_TR_Y, WDW_BL_X, WDW_BLY
 *     Set up for full screen ( default condition )
 *
 */

typedef enum {TOUCH_NOTOUCH,TOUCH_DEBOUNCE,TOUCH_PRESENT}
TOUCH_STATE;

#define TSC_ADDR  0x82     // (i2c address for touchscreen)<<1

// CHIP_ID register
#define CHIP_ID       0x00  // 16 bit register
// ID_VER register
#define ID_VER        0x02  // 8 bit register
// SYS_CTRL1 Reset control register
#define SYS_CTRL1     0x03
#define SOFT_RESET 0x02
```

```
// SYS_CTRL2 clock control register
#define SYS_CTRL2     0x04
#define ADC_OFF    0x01
#define TSC_OFF    0x02
#define GPIO_OFF   0x04

// ADC_CTRL registers
#define ADC_CTRL1     0x20
#define ADC_CTRL2     0x21

// Interrupt control, enable and status registers
#define INT_CTRL      0x09
#define INT_EN        0x0A
#define INT_STA       0x0B

// TSC_CTRL touchscreen control register
#define TSC_CTRL             0x40
#define TSC_TOUCH_DET    0x80 // 1 when touch detected else 0
#define TSC_TRACK_MASK   0x70 // 0 => no window tracking
#define TSC_OP_MOD_MASK  0x0E // 0 => XYZ acquisition
#define TSC_EN_MASK      0x01 // enable touchscreen

// TSC_CFG touchscreen config register
#define TSC_CFG       0x41
#define TSC_AVG4    0x80
#define TSC_DLY_1MS 0x20
#define TSC_STL_1MS 0x03

#define TSC_I_DRIVE   0x58
#define MAMP_50      0x01

// FIFO_TH touchscreen fifo threshold register
#define FIFO_TH       0x4A

// TSC_DATA_X X data register
#define TSC_DATA_X    0x4D

// TSC_DATA_Y  Y data register
#define TSC_DATA_Y    0x4F

// FIFO_STA  touchscreen fifo control-status register
#define FIFO_STA      0x4B
#define FIFO_OFLOW 0x80
#define FIFO_FULL  0x40
#define FIFO_EMPTY 0x20
#define FIFO_TRGD  0x10
```

```
#define FIFO_RESET 0x01

// TSC_DATA touchscreen data register
#define TSC_DATA      0xD7

// GPIO_AF -- GPIO Alternate FunctionRegister
#define GPIO_AF       0x17

DigitalOut led4(LED4);

i2c_t touch_ctrl;  // i2c interface struct for touch screen

static bool touch_present(void);
static void touch_compute_params(void);

static char tsc_reset[2]={SYS_CTRL1, SOFT_RESET};
static char clox_on[2]={SYS_CTRL2,0x00};
static char adc_ctrl1[2]={ADC_CTRL1,0x49};
static char adc_ctrl2[2]={ADC_CTRL2,0x01};
static char gpio_af[2]={GPIO_AF,0};
static char fifo_clear[2]={FIFO_STA,0x01};
static char fifo_operate[2]={FIFO_STA,0x00};
static char touch_int_en[2]={INT_EN,0x02};//enable FIFO_TH int
static char clr_intrupts[2]={INT_STA,0xFF};
static char ena_intrupt[2]={INT_CTRL,0x02};
static char tsc_cfg[2]={TSC_CFG,( TSC_AVG4
                                | TSC_DLY_1MS
                                | TSC_STL_1MS )};
static char tsc_enable[2]={TSC_CTRL,3};
static char tsc_ctrl=TSC_CTRL;
static char tsc_i_drive[2]={TSC_I_DRIVE,MAMP_50};
static char fifo_th[2]={FIFO_TH,1};
static char fifo_ctrl_sta=FIFO_STA;
static char tsc_data=TSC_DATA;

static int touch_audio_freq = 1000;
static int touch_audio_amplitude = 0;
static char touch_status=0; // result of reading TSC_CTR
static char fifo_status=0;  // result of reading FIFO_CTRL_STA

short touch_x,touch_y;

// used by state machine that debounces touch detection
TOUCH_STATE touch_state=TOUCH_NOTOUCH;
#define GOOD_TOUCH_COUNT 8
#define NO_TOUCH_COUNT 8
```

```
bool touch_init(void)
{
    char chipid[2];

    i2c_init(&touch_ctrl,p28,p27);
    i2c_frequency(&touch_ctrl,100000);
    wait_ms(1);

    // read chip id
    i2c_write(&touch_ctrl,TSC_ADDR,CHIP_ID,2,0);
    i2c_read(&touch_ctrl,TSC_ADDR,chipid,2,1);
    wait_ms(1);

    // reset touch screen chip
    i2c_write(&touch_ctrl,TSC_ADDR,tsc_reset,2,1);
    wait_ms(5);

    i2c_write(&touch_ctrl,TSC_ADDR,tsc_i_drive,2,1);
    wait_ms(1);

    // turn on ADC and TSC clocks
    i2c_write(&touch_ctrl,TSC_ADDR,clox_on,2,1);
    wait_ms(3);

    // enable touch interrupt
    i2c_write(&touch_ctrl,TSC_ADDR,touch_int_en,2,1);
    wait_ms(1);

    // 80 clock cycles for ADC conv, 12 bit ADC, internal ref
    i2c_write(&touch_ctrl,TSC_ADDR,adc_ctrl1,2,1);
    wait_ms(2);

    // ADC clock = 3.25 MHz
    i2c_write(&touch_ctrl,TSC_ADDR,adc_ctrl2,2,1);
    wait_ms(1);

    // 4 sample averaging and 1ms delays
    i2c_write(&touch_ctrl,TSC_ADDR,tsc_cfg,2,1);
    wait_ms(1);

    // gpio alt function register to 0
    i2c_write(&touch_ctrl,TSC_ADDR,gpio_af,2,1);
    wait_ms(1);

    // FIFO threshold not zero
```

```
    i2c_write(&touch_ctrl,TSC_ADDR,fifo_th,2,1);
    wait_ms(1);

    // FIFO Reset
    i2c_write(&touch_ctrl,TSC_ADDR,fifo_clear,2,1);
    wait_ms(1);

    // FIFO out of reset
    i2c_write(&touch_ctrl,TSC_ADDR,fifo_operate,2,1);
    wait_ms(1);

    // enable touchscreen, no window tracking, x,y mode
    i2c_write(&touch_ctrl,TSC_ADDR,tsc_enable,2,1);
    wait_ms(1);

    i2c_write(&touch_ctrl,TSC_ADDR,clr_intrupts,2,1);
    wait_ms(1);

    i2c_write(&touch_ctrl,TSC_ADDR,ena_intrupt,2,1);
    wait_ms(1);

    touch_state = TOUCH_NOTOUCH;
    return true;
}

bool touch_debounce(void)
{
    static int debounce_count=0;

    switch (touch_state) {
    case TOUCH_NOTOUCH:
        if (touch_present()) {
            debounce_count=0;
            touch_state = TOUCH_DEBOUNCE;
        }
        break;

    case TOUCH_DEBOUNCE:
        if (touch_present()) {
            if (++debounce_count > GOOD_TOUCH_COUNT) {
                touch_state = TOUCH_PRESENT;
            }
        } else {
            touch_state = TOUCH_NOTOUCH;
        }
        break;
```

```
    case TOUCH_PRESENT:
        if (touch_present()) {
            touch_compute_params();
            return true;
        } else {
            touch_state = TOUCH_NOTOUCH;
        }
        break;

    }
    return false;
}

int touch_frequency(void)
{
    return touch_audio_freq;
}

int touch_amplitude(void)
{
    return touch_audio_amplitude;
}
void touch_compute_params(void)
{
    if(0>touch_get_xy(&touch_x,&touch_y)) return;
    touch_audio_freq = TOUCH_MIN_FREQUENCY
        + (TOUCH_MAX_FREQUENCY - TOUCH_MIN_FREQUENCY)
        * touch_y/0xFFF;
    touch_audio_amplitude = TOUCH_MAX_AMPLITUDE*touch_x/0xFFF;
//    debug_hexshort((short)touch_audio_freq);
//    debug_putch(',');
//    debug_hexshort((short)touch_audio_amplitude);
//    debug_crlf();
}

bool touch_present(void)
{
    i2c_init(&touch_ctrl,p28,p27); // sync i2c routines
    i2c_frequency(&touch_ctrl,100000); // 100kHz clock

    i2c_write(&touch_ctrl,TSC_ADDR,&tsc_ctrl,1,0);
    i2c_read(&touch_ctrl,TSC_ADDR,&touch_status,1,1);

    if ((touch_status & TSC_EN_MASK)==0) {  // i2c error
        led4=1;                             // disables screen
```

```
        wait_ms(10);
        touch_init();                       // re-init fixes
        return false;
    } else if ((touch_status & TSC_TOUCH_DET)>0) {
        return true;
    }
    return false;
}

int touch_get_xy(short *x,short *y)
{
    unsigned char packed_xy[3];

    if(i2c_write(&touch_ctrl,TSC_ADDR,&fifo_ctrl_sta,1,0)<0)
        return NIL;
    if(i2c_read(&touch_ctrl,TSC_ADDR,&fifo_status,1,1)<0)
        return NIL;
    while ((fifo_status & FIFO_EMPTY)!=FIFO_EMPTY ) {
        if (i2c_write(&touch_ctrl,TSC_ADDR,&tsc_data,1,0) < 0)
            return NIL;
        if(i2c_read(&touch_ctrl,TSC_ADDR,(char *)packed_xy,
3,1)<0)
            return NIL;
        if(i2c_write(&touch_ctrl,TSC_ADDR,&fifo_ctrl_sta,1,0)<0)
            return NIL;
        if(i2c_read(&touch_ctrl,TSC_ADDR,&fifo_status,1,1)<0)
            return NIL;
    }
    if(i2c_write(&touch_ctrl,TSC_ADDR,clr_intrupts,2,1)<0)
        return NIL;
    *x = (short)(packed_xy[0]<<4)
       | (short)((packed_xy[1] & 0xF0)>>4);
    *y = 0xFFF -((short)((packed_xy[1] & 0x0F)<<8)
               | (short)(packed_xy[2]));
    // 0xFFF or 0x000 is some kind of glitch
    if (*x==0xFFF && *y==0xFFF) return NIL;
    if (*x==0 && *y==0) return NIL;
//    debug_hexshort(*x);
//    debug_putch(',');
//    debug_hexshort(*y);
//    debug_crlf();
    return 0;
}
```

```
#ifndef DMA_H
#define DMA_H

#define DMA_BUFSIZE 512
#define DAC_POWER_MODE (1<<16)

void dma_init(void);
void dma_enable(void);
void dma_disable(void);
int *dma_get_bufptr(int bufno);
void TC0_callback(void);
void ERR0_callback(void);
void TC1_callback(void);
void ERR1_callback(void);


#endif
```

```
/*
 * dma.cpp
 * Send a buffer repeatedly to the DAC using DMA.
 */
#include "mbed.h"
#include "note.h"
#include "MODDMA.h"
#include "dma.h"

int buffer[2][DMA_BUFSIZE];

AnalogOut sig(p18);  // analog output object ( uses pin 18)

MODDMA dma;
MODDMA_Config *conf0, *conf1;

int *dma_get_bufptr(int bufno)
{
    return buffer[bufno];
}

void dma_init(void) {

    // Prepare the GPDMA system for buffer0.
    conf0 = new MODDMA_Config;
    conf0
     ->channelNum    ( MODDMA::Channel_0 )
     ->srcMemAddr    ( (uint32_t) &buffer[0] )
```

```
     ->dstMemAddr    ( MODDMA::DAC )
     ->transferSize  ( 512 )
     ->transferType  ( MODDMA::m2p )
     ->dstConn       ( MODDMA::DAC )
     ->attach_tc     ( &TC0_callback )
     ->attach_err    ( &ERR0_callback )
    ; // config end


    // Prepare the GPDMA system for buffer1.
    conf1 = new MODDMA_Config;
    conf1
     ->channelNum    ( MODDMA::Channel_1 )
     ->srcMemAddr    ( (uint32_t) &buffer[1] )
     ->dstMemAddr    ( MODDMA::DAC )
     ->transferSize  ( 512 )
     ->transferType  ( MODDMA::m2p )
     ->dstConn       ( MODDMA::DAC )
     ->attach_tc     ( &TC1_callback )
     ->attach_err    ( &ERR1_callback )
    ; // config end


//
// By default, the Mbed library sets the PCLK_DAC clock value
// to 24MHz. One wave cycle in each buffer is DMA_BUFSIZE
// (512) points long. The sample rate of the output is to be
// WAVE_SAMPLE_RATE (22050) samples per second, regardless of.
// wave frequency.  So the DACCNTVAL will be 24000000/22050
// or 1088.
//

    LPC_DAC->DACCNTVAL = 1088; // for 22050 sample rate

//
// the wave templates will be one dma buffer long, so the
// lowest frequency that can be produced is
//  WAVE_SAMPLE_RATE/DMA_BUFSIZE = 22050/512 = 43 Hz
// the highest frequency wave that can be produced is
// roughly half the sample rate, or 11025 Hz.
//
}

void dma_enable(void)
{
    // Prepare first configuration.
```

```
    if (!dma.Prepare( conf0 )) {
        error("Doh!");
    }

    // Begin (enable DMA and counter). Note, don't enable
    // DBLBUF_ENA as we are using DMA double buffering.
    LPC_DAC->DACCTRL |= (3UL << 2);
}

void dma_disable(void)
{

    // Finish the DMA cycle by shutting down the channel.
    dma.Disable((MODDMA::CHANNELS)conf0->channelNum());
    dma.Disable((MODDMA::CHANNELS)conf1->channelNum());

}

// Configuration callback on TC
void TC0_callback(void) {

    dma.Disable((MODDMA::CHANNELS)conf0->channelNum());

    if (note_active()) {
        // Notify note.cpp that it is time to refill buffer[0]
        note_set_bufno(0);

        // Swap to buffer1
        dma.Prepare( conf1 );

        // Clear DMA IRQ flags.
        if (dma.irqType() == MODDMA::TcIrq) dma.clearTcIrq();
    } else {
        dma.Disable((MODDMA::CHANNELS)conf1->channelNum());
    }
}

// Configuration callback on Error
void ERR0_callback(void) {
    error("Oh no! My Mbed EXPLODED!");
}

// Configuration callback on TC
void TC1_callback(void) {

    // Finish the DMA cycle by shutting down the channel.
```

```
    dma.Disable((MODDMA::CHANNELS)conf1->channelNum());

    if (note_active()) {
        // Notify note.cpp that it is time to refill buffer[1]
        note_set_bufno(1);

        // Swap to buffer0
        dma.Prepare( conf0 );

        // Clear DMA IRQ flags.
        if (dma.irqType() == MODDMA::TcIrq) dma.clearTcIrq();
    } else {
        dma.Disable((MODDMA::CHANNELS)conf0->channelNum());
    }
}

// Configuration callback on Error
void ERR1_callback(void) {
    error("Oh no! My Mbed EXPLODED!");
}
```

```
#ifndef WAVE_H
#define WAVE_H

#define WAVE_TYPE_FIRST    0
#define WAVE_TYPE_SAW      0
#define WAVE_TYPE_TRIANGLE 1
#define WAVE_TYPE_SQUARE   2
#define WAVE_TYPE_LAST     2
#define WAVE_TYPE_DEFAULT WAVE_TYPE_SAW
#define WAVE_SAMPLE_RATE   22050

void wave_init(void);
void wave_reset(void);
void wave_update(void);
int wave_nextval(unsigned freq);
bool wave_type_changed(void);
void wave_type_incr(void);
#endif
```

```
/*
 * wave.cpp -- wave templates for use in Digital
 *    Theremin demo program
 */
#include "mbed.h"
#include "dma.h"
#include "jswitch.h"
#include "wave.h"

// Default wave type is the SAW
static int wavetype = WAVE_TYPE_DEFAULT;
// Next wave type requested
static bool next_wavetype=false;
// The wave template buffer size is DMA_BUFSIZE
static int waveform[1+WAVE_TYPE_LAST - WAVE_TYPE_FIRST]
[DMA_BUFSIZE];
// Accumulated phase
static unsigned accum_phi = 0;

/*
 * the phase accumulator ranges between 0 and 99999.
 * corresponding to 0 and 2*pi radians.  The frequency
 * supplied to the wave_nextval() function determines the
 * magnitude of the  change in the phase accumulator value,
 * according to the formula:
 *       delta-phase = 100000*frequency/WAVE_SAMPLE_RATE
 * The next value of phase_accumulator is:
```

```
 *       (phase_accumulator + delta-phase) % 100000
 * phase_accumulator is converted to wave table index using
 * the formula:
 *        index = (DMA_BUFSIZE - 1)*phase_accumulator/100000
 */

void wave_init(void)
{
    int j;

    accum_phi = 0;
    wavetype = WAVE_TYPE_DEFAULT;

    for (j=0;j<DMA_BUFSIZE;j++) {
        waveform[WAVE_TYPE_SAW][j] = j;
    }

    for (j=0;j<DMA_BUFSIZE;j++) {
        if (j<DMA_BUFSIZE/2) {
            waveform[WAVE_TYPE_SQUARE][j] = DMA_BUFSIZE;
        } else {
            waveform[WAVE_TYPE_SQUARE][j] = 0;
        }
    }

    for (j=0;j<DMA_BUFSIZE;j++) {
        if (j<DMA_BUFSIZE/2) {
            waveform[WAVE_TYPE_TRIANGLE][j] = 2*j;
        } else {
            waveform[WAVE_TYPE_TRIANGLE][j]
                =DMA_BUFSIZE-2*(j-DMA_BUFSIZE/2);
        }
    }
}

void wave_reset(void)
{
    accum_phi=0;
}
/*
 * wave_nextval
 *       delta-phase = 100000*frequency/WAVE_SAMPLE_RATE
 * The next value of phase_accumulator is:
 *       (phase_accumulator + delta-phase) % 100000
 * phase_accumulator is converted to wave table index using
 * the formula:
```

```
 *        index = DMA_BUFSIZE * phase_accumulator / 100000
 */
int wave_nextval(unsigned freq)
{
    unsigned delta_phi,index;

    delta_phi= 100000*freq/WAVE_SAMPLE_RATE;
    accum_phi = (accum_phi + delta_phi)%100000;
    index = ((DMA_BUFSIZE - 1) * accum_phi)/100000;
    return waveform[wavetype][index];
}

void wave_update(void)
{
    unsigned char js_val;

    js_val = js_read(JS_CENTER);
    if ((js_val & JS_CENTER)==JS_CENTER) {
        next_wavetype = true;
    }
}

bool wave_type_changed(void)
{
    return next_wavetype;
}

void wave_type_incr(void)
{
    next_wavetype = false;
    wavetype = 1+wavetype;
    if (wavetype > WAVE_TYPE_LAST) {
        wavetype = WAVE_TYPE_FIRST;
    }
}
```

```
#ifndef ENVLP_H
#define ENVLP_H
/*
 * envlp.h -- Functions which help note.cpp produce
 * an ADSR (Attack,Decay,Sustain,Release) envelope
 */
#define ENVLP_MAX (0x1FF)

void envlp_init(void);
void envlp_update(void);
int envlp_get_attack_bufs(void);
int envlp_get_decay_bufs(void);
int envlp_get_release_delta(void);

#endif

/*
 * envlp.c -- manage the note's envelope state machine
 */
#include "mbed.h"
#include "debug.h"
#include "touch.h"
#include "jswitch.h"
#include "envlp.h"
/*
 * sound envelope follows ADSR pattern
 * (Attack, Decay, Sustain, Release)
 * three parameters below control the envelope shape
 * It should be easy to consider these default values
 * and modify the code below to set the variables
 * envlp_attack_bufs, envlp_release_bufs,
 * and envlp_release_delta dynamically,
 * depending upon external inputs
 */
#define ATTACK_BUFFERS_MIN  2
#define ATTACK_BUFFERS_MAX 20
#define ATTACK_BUFFERS 5     // attack buffers
#define DECAY_BUFFERS_MIN 1
#define DECAY_BUFFERS_MAX 40
#define DECAY_BUFFERS  ATTACK_BUFFERS/2 // decay buffers
#define RELEASE_DELTA_MIN 1
#define RELEASE_DELTA_MAX 128
#define RELEASE_DELTA  8    // inverse release duration param

/*
 * local envelope functions
```

```
 */
static void envlp_set_attack_bufs(void);
static void envlp_set_decay_bufs(void);
static void envlp_set_release_delta(void);

/*
 * local envelop veriables
 */
static int envlp_attack_bufs;
static int envlp_decay_bufs;
static int envlp_release_delta;

void envlp_init(void)
{
    envlp_attack_bufs=ATTACK_BUFFERS;
    envlp_decay_bufs=DECAY_BUFFERS;
    envlp_release_delta=RELEASE_DELTA;
}

void envlp_update(void)
{
    envlp_set_attack_bufs();
    envlp_set_decay_bufs();
    envlp_set_release_delta();
}

void envlp_set_attack_bufs()
{
    unsigned char js_val;

    js_val = js_read((JS_UP | JS_DOWN));
    if (js_val & JS_UP) {
        if (envlp_attack_bufs < ATTACK_BUFFERS_MAX) {
            envlp_attack_bufs++;
        }
    } else if(js_val & JS_DOWN) {
        if (envlp_attack_bufs > ATTACK_BUFFERS_MIN) {
            envlp_attack_bufs--;
        }
    }
}

int envlp_get_attack_bufs(void)
{
    return envlp_attack_bufs;
}
```

```
void envlp_set_decay_bufs(void)
{
    if (envlp_attack_bufs>1) {
        envlp_decay_bufs = envlp_attack_bufs/2;
    } else {
        envlp_decay_bufs = 1;
    }
}

int envlp_get_decay_bufs(void)
{
    return envlp_decay_bufs;
}

void envlp_set_release_delta(void)
{
    unsigned char js_val;

    js_val = js_read((JS_LEFT | JS_RIGHT));
    if (js_val & JS_LEFT) {
        if (envlp_release_delta < RELEASE_DELTA_MAX) {
            envlp_release_delta++;
        }
    } else if(js_val & JS_RIGHT) {
        if (envlp_release_delta > RELEASE_DELTA_MIN) {
            envlp_release_delta--;
        }
    }
}

int envlp_get_release_delta(void)
{
    return envlp_release_delta;
}
```

```
#ifndef NOTE_H
#define NOTE_H
/*
 * note.h
 *    Start, update, release or end note, and fill dma buffers
for
 *    one note at a time.
 */
#define NIL (-1)

void note_init(void);
void note_update(void);
bool note_active(void);
void note_set_bufno(int bufno);
#endif

/*
 *  note.cpp -- manage production of notes in response to
 *     touch screen input
 */
#include "mbed.h"
#include "InterruptIn.h"
#include "debug.h"
#include "dma.h"
#include "envlp.h"
#include "wave.h"
#include "touch.h"
#include "jswitch.h"
#include "note.h"

DigitalOut led2(LED2);   // lit during NOTE_RELEASE state

typedef enum
{NOTE_ATTACK,NOTE_DECAY,NOTE_SUSTAIN,NOTE_RELEASE,NOTE_OFF}
NOTE_STATE;

static void note_start(void);
static void note_state_machine(void);
static void note_fill_buf(void);
static void note_attack(void);
static bool note_attack_done(void);
static bool note_decay_done(void);
static void note_release(void);
static bool note_release_done(void);
static bool note_released(void);
static void note_end(void);
```

```
static unsigned note_freq=0;        // frequency in Hz
static NOTE_STATE note_state=NOTE_OFF;
static int note_bufno= NIL;       // NIL or index of dma buffer
static int note_attack_bufcount;
static int note_attack_delta;
static int note_attack_bufs;
static int note_decay_bufs;
static int note_decay_bufcount;
static int note_decay_delta;
static int note_release_delta;
static int note_first_bufval;
static int note_last_bufval;
static int note_release_numerator;
static int note_release_denominator;

void note_init(void)
{
    note_freq = 0;
    note_state = NOTE_OFF;
    note_bufno = NIL;
    note_attack_bufcount=0;
    note_decay_bufcount=0;
    dma_init();
}

void note_start(void)
{
    if (wave_type_changed()) {
        wave_type_incr();
    }
    wave_reset();
    note_freq = touch_frequency();
    note_attack_bufs=envlp_get_attack_bufs();
    debug_hexshort((short)note_attack_bufs);
    debug_putch(' ');
    note_decay_bufs=envlp_get_decay_bufs();
    note_release_delta = envlp_get_release_delta();
    note_release_numerator = ENVLP_MAX - note_release_delta;
    note_release_denominator = ENVLP_MAX;
    debug_hexshort((short)note_release_numerator);
    debug_crlf();

    // fill first two buffers
    note_attack();
    note_set_bufno(0);
```

```
    note_state_machine();
    note_fill_buf();
    note_set_bufno(1);
    note_state_machine();
    note_fill_buf();
    dma_enable();
}

void note_end(void)
{
    dma_disable();
    note_state=NOTE_OFF;
    note_bufno=NIL;
    led2 = 0;
}

/*
 * note_update
 * update the frequency and attenuation factor for the
 * current note.   If the current envelope first and last
 * add up to less than 2, end the note.
 * otherwise check to see if it is time to fill a dma buffer.
 * if so fill the one specified by note_fillbuf.
 */

void note_update(void)
{
    // execute note state machine
    note_state_machine();

    // if dma buffer sent, refill it
    if(note_bufno!=NIL) {
       note_fill_buf();
    }

    // handle presence or absence of touch
    if (touch_debounce()) {
        wait_ms(10);  // wait 10 msec if touch present
        if (note_released() || !note_active()) {
            note_start();
        }
    } else {
        if (note_active()) {
            if (!note_released()) {
                note_release();
            }
```

```
        }
    }
}

void note_state_machine(void)
{
     // execute state machine that manages envelope

    switch (note_state) {
    case NOTE_ATTACK:
        if (note_attack_done()) {
            note_state = NOTE_DECAY;
        }
        note_freq = touch_frequency();
        break;

    case NOTE_DECAY:
        if(note_decay_done()) {
            note_state = NOTE_SUSTAIN;
        }
        note_freq = touch_frequency();
        break;

    case NOTE_SUSTAIN:
        note_freq = touch_frequency();
        break;

    case NOTE_RELEASE:
        if (note_release_done()) {
            note_end();
            break;
        }
        break;
    case NOTE_OFF:
    default:
        break;
    }
}

/*
 * note_fill_buf
 *    Use the first and last buffer envelope values, and the
 *    wave values returned by wave_nextval() to compute the
 *    envelop-modified wave values, and then apply the changes
 *    necessary to output the values to the DAC data register.
 */
```

```
void note_fill_buf(void)
{
    int j,start_env,end_env,env_val,wave_val,buf_val;
    int *bufptr=NULL;

    bufptr = dma_get_bufptr(note_bufno);
    start_env = note_first_bufval;
    end_env = note_last_bufval;

    for (j=0;j<DMA_BUFSIZE;j++) {
        env_val=start_env+(end_env - start_env)*j/DMA_BUFSIZE;
        wave_val = wave_nextval(note_freq);
        buf_val = wave_val * env_val / ENVLP_MAX;
        buf_val=touch_amplitude()*buf_val/TOUCH_MAX_AMPLITUDE;
        bufptr[j]= DAC_POWER_MODE | ((buf_val << 6) & 0xFFC0);
    }
    note_bufno = NIL;  // buffer is filled
    switch (note_state) {
    case NOTE_ATTACK:
        note_attack_bufcount++;
        break;
    case NOTE_DECAY:
        note_decay_bufcount++;
        break;
    case NOTE_RELEASE:
        note_first_bufval = note_last_bufval;
        note_last_bufval =
            note_first_bufval * note_release_numerator
            / note_release_denominator;
        break;
    case NOTE_SUSTAIN:
    case NOTE_OFF:
    default:
        ;
    }
}
/*
 * note_release
 *    touch has been lifted, note begins to decay
 */
void note_release(void)
{
    led2 = 1;
    note_state = NOTE_RELEASE;
}
/*
```

```
 * note_released
 *    return true if note has been released
 *    but not yet ended
 */
bool note_released(void)
{
    return (note_state == NOTE_RELEASE);
}
bool note_active(void)
{
    return (note_state < NOTE_OFF);
}
/*
 ** note_set_bufno
 *     Set the index of the dma buffer to fill.
 */
void note_set_bufno(int bufno)
{
    note_bufno = bufno;
}

void note_attack(void)
{
    note_state = NOTE_ATTACK;
    note_attack_bufcount = 0;
    note_attack_delta = ENVLP_MAX / note_attack_bufs;
    note_first_bufval=note_attack_bufcount*note_attack_delta;
    note_last_bufval=note_first_bufval + note_attack_delta;
}

bool note_attack_done(void)
{
    if (note_attack_bufcount >= note_attack_bufs) {
        note_decay_bufcount = 0;
        note_decay_delta = (ENVLP_MAX >> 1) / note_decay_bufs;
        note_first_bufval = note_last_bufval;
        return true;
    } else {
     note_first_bufval=note_attack_bufcount*note_attack_delta;
        note_last_bufval=note_first_bufval+note_attack_delta;
    }
    return false;
}

bool note_decay_done(void)
{
```

```
    if (note_decay_bufcount >= note_decay_bufs) {
        note_first_bufval = note_last_bufval;
        return true;
    } else {
        note_last_bufval=note_first_bufval - note_decay_delta;
        if (note_last_bufval < 0) {
            note_last_bufval=0;
        }
    }
    return false;
}

bool note_release_done(void)
{
    if (note_first_bufval < 25) return true;
    return false;
}
```

BIBLIOGRAPHY

1. The Unified Modeling Language User Guide, Booch, Rumbaugh, Jacobson, 1999, Addison-Wesley.

2. Object Oriented Software Engineering, Ivar Jacobson,1992, ACM Press,Addison-Wesley.

3. Managing Software Requirements: A Unified Approach, Leffingwell,Widrig, 1999, Addison-Wesley.

4. Software Cost Estimation with COCOMO II,Barry Boehm,2000,Prentice-Hall.

5. Function Point Analysis,Garmus,Herron,2000,Addison-Wesley.

6. Object Oriented Design, Grady Booch, 1991,The Benjamin/Cummings Publishing Company, Inc.

7. Software Testing Techniques,Beizer,1990,Van Norstrand Reinhold.