HOW THE PROS DEVELOP EMBEDDED SOFTWARE

A no-nonsense guide for developers

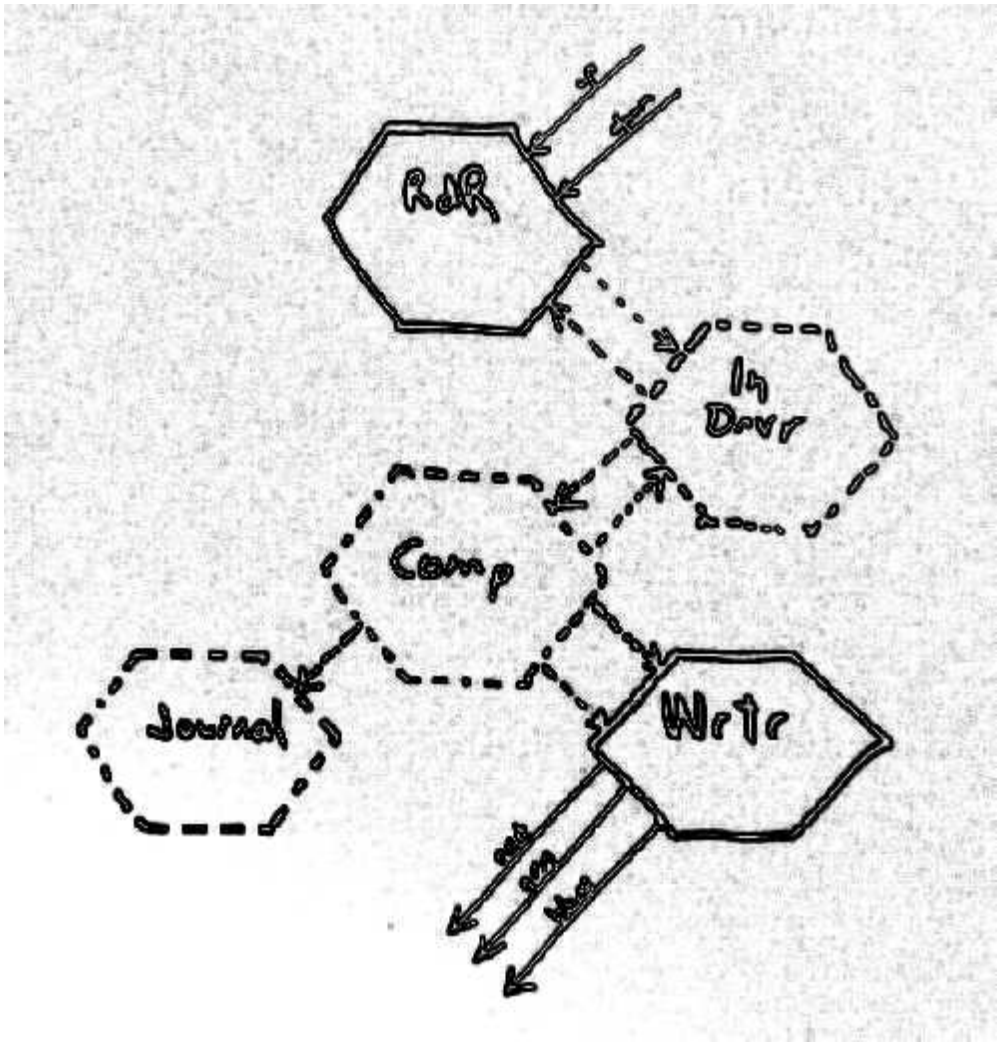


Table of Contents

INTRODUCTION

Too many books on embedded software development deliver conceptually elegant models of the development process, but lack the down-to-earth details of how to actually get the job done.

This no-nonsense guide fills in those details by showing an actual project from start to finish, and all of the methods and documents created along the way. This information comes directly from the author's 20 years of experience developing embedded software for electronic products made by top U.S. corporations

This guide walks the reader through the creation of a typical embedded application, a voice controlled synthesizer, from requirements analysis through validation.

It is intended that a programmer with no previous embedded experience can read this book, and feel neither lost nor out of place in his first embedded project. The new embedded developer may use the methods in this book as a starting point for his own collection of techniques.

The sample application is called a voice substitution device. It measures the fundamental frequency, if any, in an input signal; and substitutes a synthesized waveform of the same frequency in the output. Hopefully, this example is challenging enough to illustrate most common embedded development practices.

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.

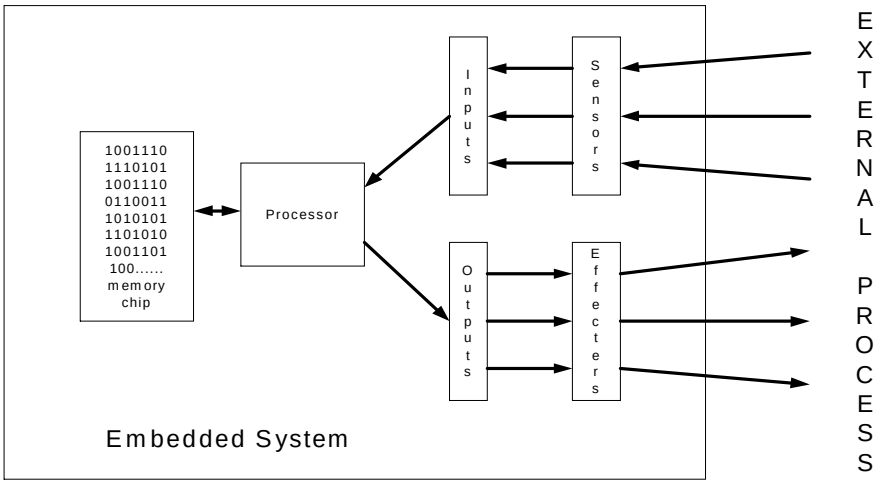


FIGURE 1. Embedded System

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. 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 composed of some hardware and some virtual objects.

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. This conceptual map has both a material and a mental form.

MATERIAL CONCEPTUAL MAP

The material form of the conceptual map is a collection of data sheets, analysis and design documents, interviews, and meeting minutes which bear on the project. These documents are usually kept organized and readily accessible to each member of the team.

Material form of the Conceptual Map

MENTAL CONCEPTUAL MAP

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

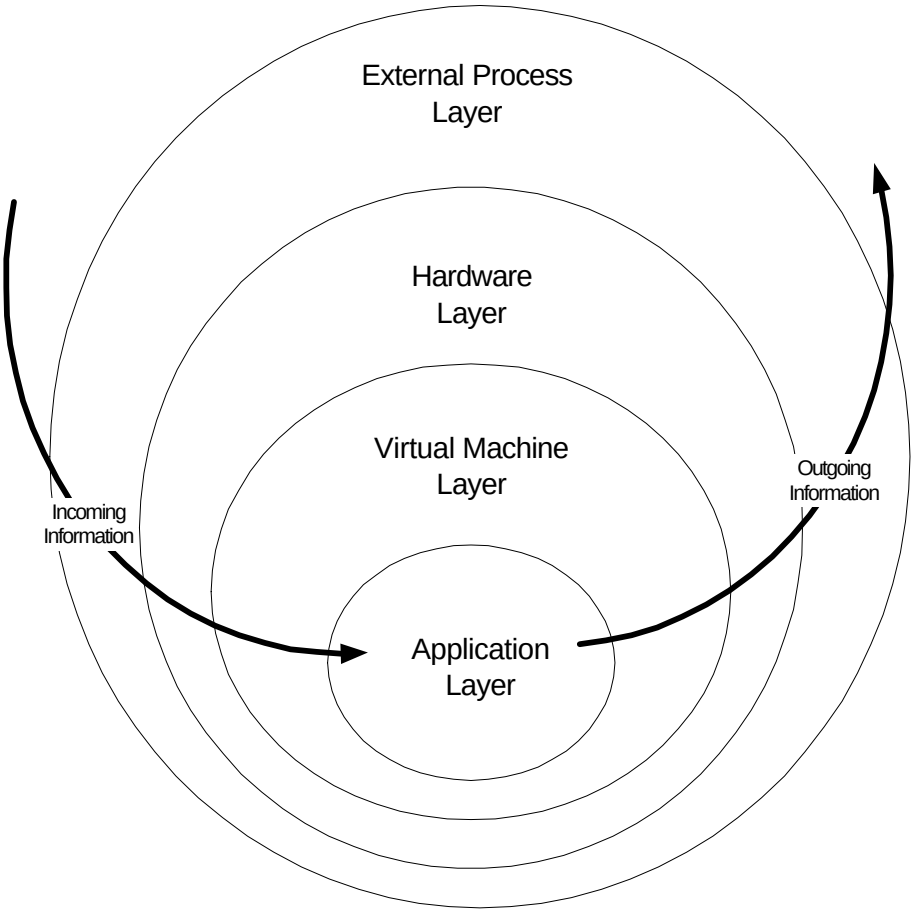
Pessimistic Impression of a "Mental Conceptual Map"

The dreaded "learning curve" of an embedded project is the time necessary for a new team member to create, from conversations and from the project's material conceptual map, a mental conceptual 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 real external world, and the embedded software which interacts with it.

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



In this diagram, the subdivisions of the conceptual map relate to the flow of information in the embedded system. Information flows in from the external process layer through the hardware layer, is transformed by the virtual machine layer, and processed by the application layer. Response information is sent back out via the virtual machine layer, through the hardware layer, into 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 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 I/O drivers supplied with the operating system or written separately. Some of the concepts and documents for this layer are supplied by language and operating system vendors, and some are created by the developers.

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 access to a well organized collection of system documents, and if they take the time to read and understand those documents; the project has a much higher prospect of success than if they do not.

REAL ENGINEERS

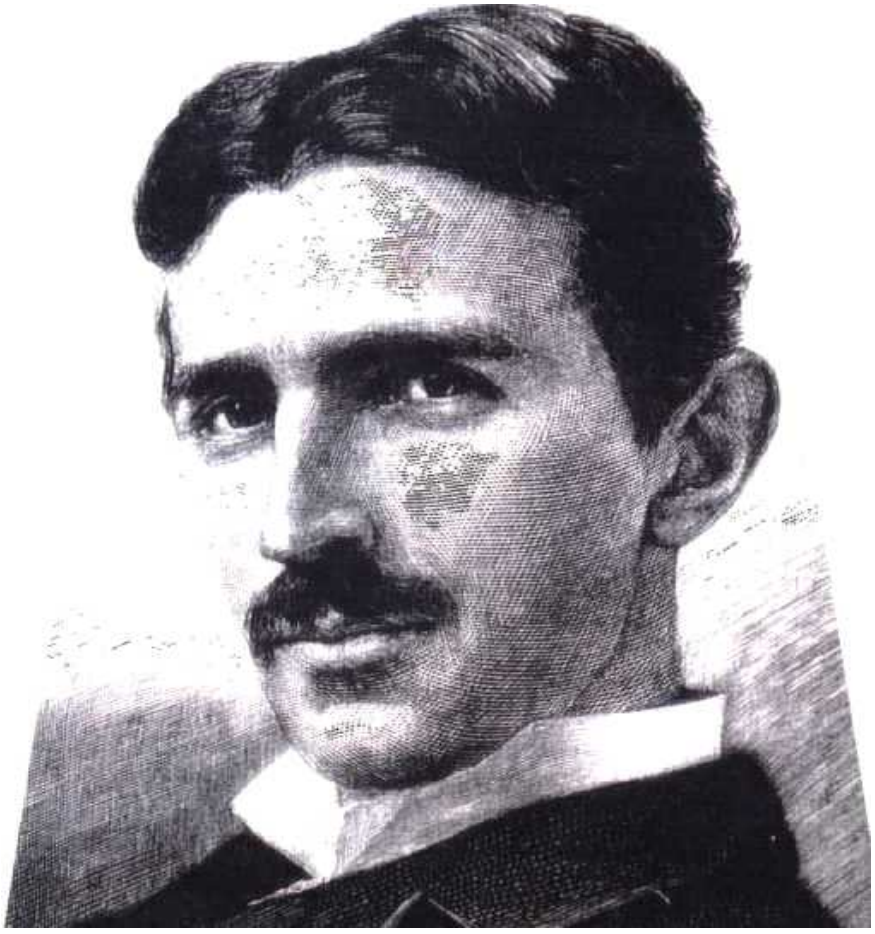


Photo of Nikola Tesla Courtesy of Tesla Memorial Society of New York
www.teslasociety.com

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

Troubles arise only when the number of callable functions exceeds a couple of hundred, and/or 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 document the conceptual map, be it in requirements, architecture, or design documents. If the reader is convinced that documenting is gilding the lily, then he will surely ignore these methods.

Nikola Tesla, arguably the Earth's greatest engineer, kept most of his plans and notes (except for patent applications) in his head. Mr. Tesla was the acknowledged master of the mental conceptual map. Perhaps that is why some of his greatest inventions weren't rediscovered until many years after his death.

Just as some engineers favor sparse documentation, others prefer rich documentation. These 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 requires her 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, and debug and test their operation. To these ends, the developer participates in the processes described below.

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.

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.

INTEGRATION
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.

VERIFICATION
Formally demonstrate that the virtual objects work properly. Revise all above as needed.

VALIDATION
Formally demonstrate that the overall embedded system meets the goals it was intended to meet. Revise all above as needed.

The processes described above will be considered in more detail in the next few sections of this book.

ANALYSIS

A better heading for this chapter might be "Learning".

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 are the technical goals of the project?
6) What is the technical environment of the project?
7) Are project resources adequate to support your efforts?
8) What is the expected schedule for the project? Is it realistic?
9) What are the skills, strengths, and weaknesses of other team members?
10) Which team members do you like? Which ones can you trust? What are their skill sets?
11) 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 couple of years 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.

Project Motivation and Support

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 find out, 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.

Once 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 the embedded system is supposed to do. You do 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 the various 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 radically different purposes, it is a good idea to use two different kinds of documentation. These are the **use case summary**, and the **formal requirements list**. In no circumstance should you attempt to communicate requirements to non-engineers using the formal requirements list.

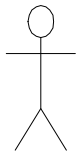
Sections below give examples of a use case summary and a formal requirements list. Then the requirements gathering techniques of reverse engineering and the requirements workshop are described.

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.

Use Case Summary

In his 1992 bestseller Object-Oriented Software Engineering, 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:

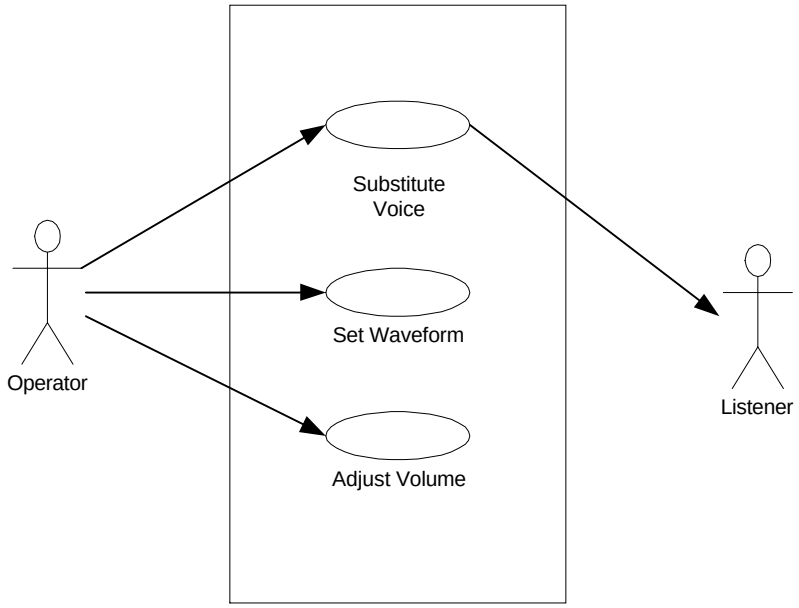

Use Case

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.

On the next page are shown the use cases for the example embedded system, a voice substitution device, which provides a way to control a synthesizer with your voice. It is a sort of voice-controlled Theramin. (The Theramin, named after the inventor, was the first electronic music synthesizer. It is controlled by varying the proximity of the operator's hands to two antennae. It is devilish hard to play.)

The voice substitution device lets you control a synthesizer with the same ease with which you control your own voice. Plus, it makes a good example for this book.

Voice Substitution Device



Actors:

Operator -- The user of the voice substitution device
Listener -- Person listening to the output. May be the same person as the operator.

Use Cases:

Substitute Voice -- The Operator vocalizes into a microphone or provides line level input which conveys an audio signal to the device. The device accepts the signal, and puts out a signal to the output jacks. Whenever a voice (fundamental frequency under 1kHz plus harmonics) is present in the input, a different voice (sine, triangle, saw, or square wave), is sent to

the output. If no voice is present in the input, the input signal is just copied to the output.

Set Waveform -- The Operator pushes a button to select one of four output waveforms: Sine, Triangle, Saw, or Square wave.

Adjust Volume -- The Operator adjusts the volume of the output by setting the potentiometer. This in turn sets the attenuation in the output stage of the Codec used by the system. This is particularly helpful if the Listener is using earphones.

That's all there is to the use cases of our example. In more complex systems, some use cases may employ others, and some use cases may be like subroutines, which are employed by many other use cases. For this reason, Dr. Jacobson allows diagrammatic linkage between the use cases.

Often it is sufficient to forego the graphical representation, and to just write the use cases in hypertext with the linkages between use cases supplied as hypertext links. Any HTML composer will work for this purpose.

Once you know what the system does, you can make a first draft of use cases for a medium sized embedded system (300-500 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 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.

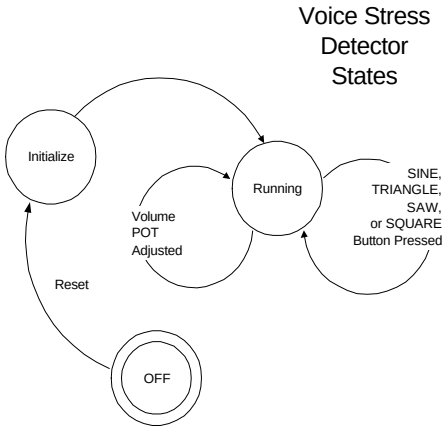
Why do you need a formal requirements list? You will refer to it often during the architecture and design processes. It will guide you in making tradeoffs. You will use it to write test plans that allow you to measure how well your work has turned out. The process of writing it will expose you to relevant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

There are many recommended formats for requirements lists. There's the 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 begins a list of formal requirements for the voice substitution device example.

1) When powered up, the voice substitution device briefly enters an initialization phase, during which it sets up the system hardware and computes waveform tables. Then it enters normal operation using full volume, and a Saw waveform as its defaults.



2) The voice stress detector accepts a microphone or line level input signal. It outputs an audio signal to the earphone or line level outputs, which consists of an echo of the input signal with voiced sounds replaced by a synthesized waveform.

3) When no voice is detected, the input signal is merely copied to the output without modification to the waveform, except that the volume may be reduced if that is requested by the potentiometer setting.

4) The voice substitution device shall monitor four hardware switches called SINE, TRIANGLE, SAW, and SQUARE, as well as the setting of a potentiometer.

5) The hardware switches shall enable the user to choose between four different output voice waveforms, sine wave, triangle wave, saw wave, or square wave.

6) The potentiometer setting shall control the volume of the output, from silence to whatever is the volume of the input.

7) Voiced portions of the signal shall have a fundamental output frequency equal to the fundamental input frequency determined by the voice detection algorithm. The frequency match shall be close enough to avoid audible disharmony between the input and the output signal over the frequency range in which the device operates.

8) The voice substitution device shall operate within the frequency range of 48Hz to 880Hz.

9) The device is expected to operate correctly with vocal input from a pre-amplified microphone. Operation from noisy, off-air or taped sources is not required.

10) The device shall be capable of recomputing a voice/unvoiced decision and voice frequency estimate two hundred times per second.

11) The device shall introduce an envelope delay of no more than 5.0 msec between the input and the output signals.

Reverse Engineering

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. For one thing, there may not have been an original requirements document. For another, 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 draw call diagrams 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 diagrams 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 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. One creates an interactive collection of virtual objects that meets the system requirements, and runs on an idealized virtual machine layer.

Early requirements models used a data-flow virtual machine[1]. Jacobson proposed a nifty, Interface-Entity-Control virtual machine for the requirements model[2].

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 will document a collection of virtual objects that are already adapted to the virtual machine layer of your target system.

[1] Structured Analysis and System Specification, Tom Demarco

[2] Object Oriented Software Engineering, Ivar Jacobson.

Safety Issues

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.

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.

Hazard 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. Suggest methods for mitigating that hazard, 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 serve to focus attention on safety issues early in the project.

Additional processes to ensure system safety are mentioned in the Design section of this book.

ARCHITECTURE

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

In this section, an **object oriented architecture** method is presented, using our example of the voice substitution 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

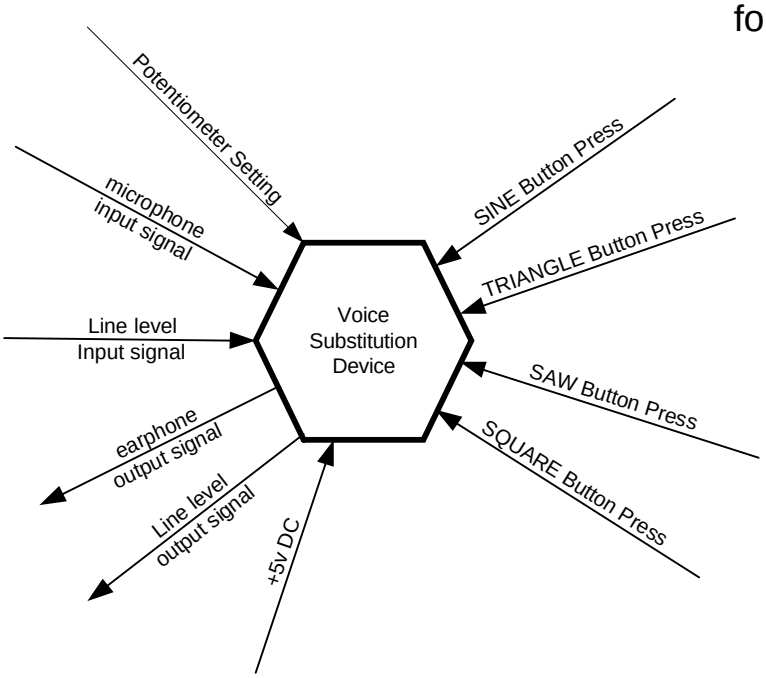
The goal of an object-oriented architecture study is to describe a collection of hardware and virtual objects, which meets the requirements produced by analysis. This section shows what goes into an object-oriented architecture study, using the example of the voice substitution device.

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.

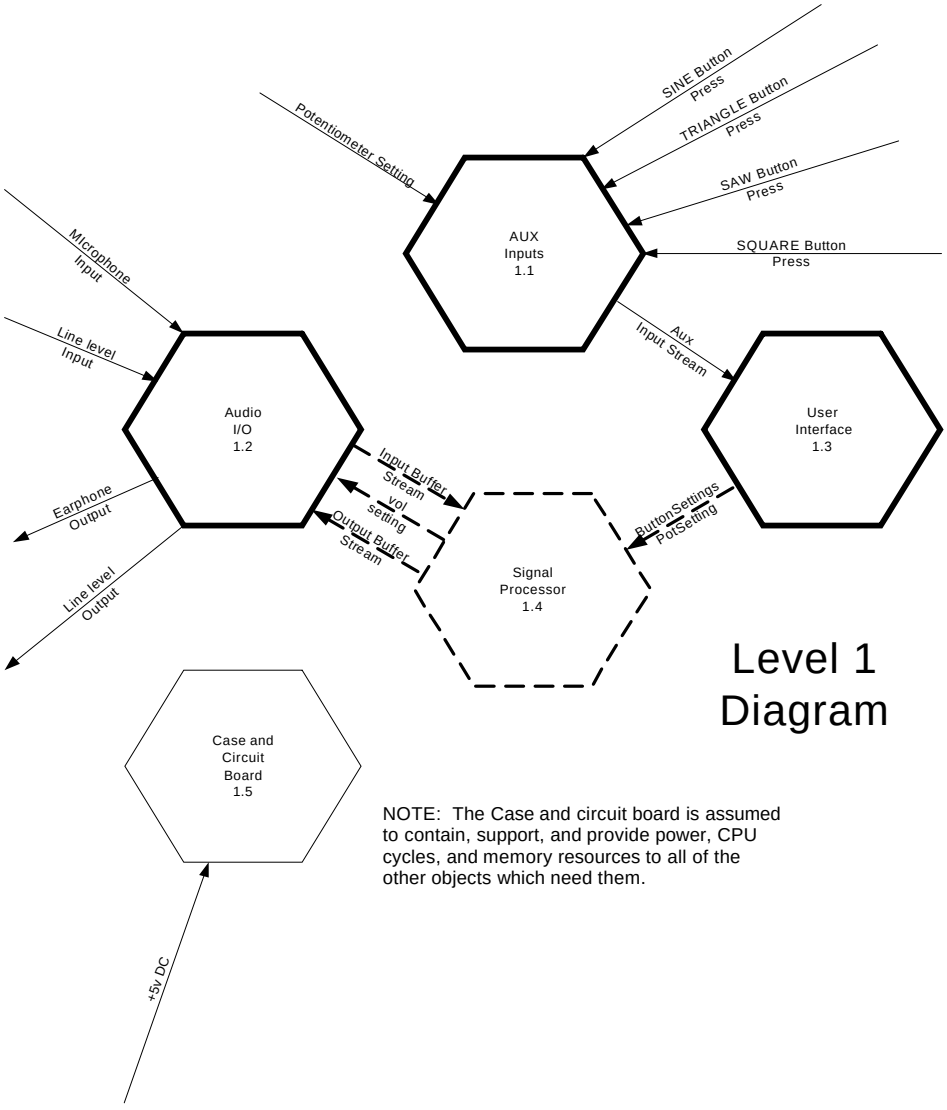
System
Context Diagram
for Voice Substitution
Device



The buttons control the waveform of the substituted voice. The potentiometer setting controls the output volume.

Next, we show 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 Diagram

NOTE: The Case and circuit board is assumed to contain, support, and provide power, CPU cycles, and memory resources to all of the other objects which need them.

1.1 Aux Inputs

The Aux Inputs object is a separate board from the rest of the system. It is a Basic Stamp2™ system with some push buttons, and a potentiometer. The software includes periodically called routines which read the button states, read the pot resistance, compose a record of those readings, and crank a bit banging state machine to pass that record to the User Interface on the main processor board.

1.2 Audio I/O

This object accepts analog input from the microphone or line level input jacks, and it converts to a stream of digital input data buffers. It also accepts a stream of digital output data buffers, and converts them to analog output on the line level and earphone output jacks.

1.3 User Interface

This object has a synchronous serial port which receives the bit stream clocked by the Aux Inputs object. It keeps track of button and potentiometer settings and supplies those values to the Signal Processor on demand.

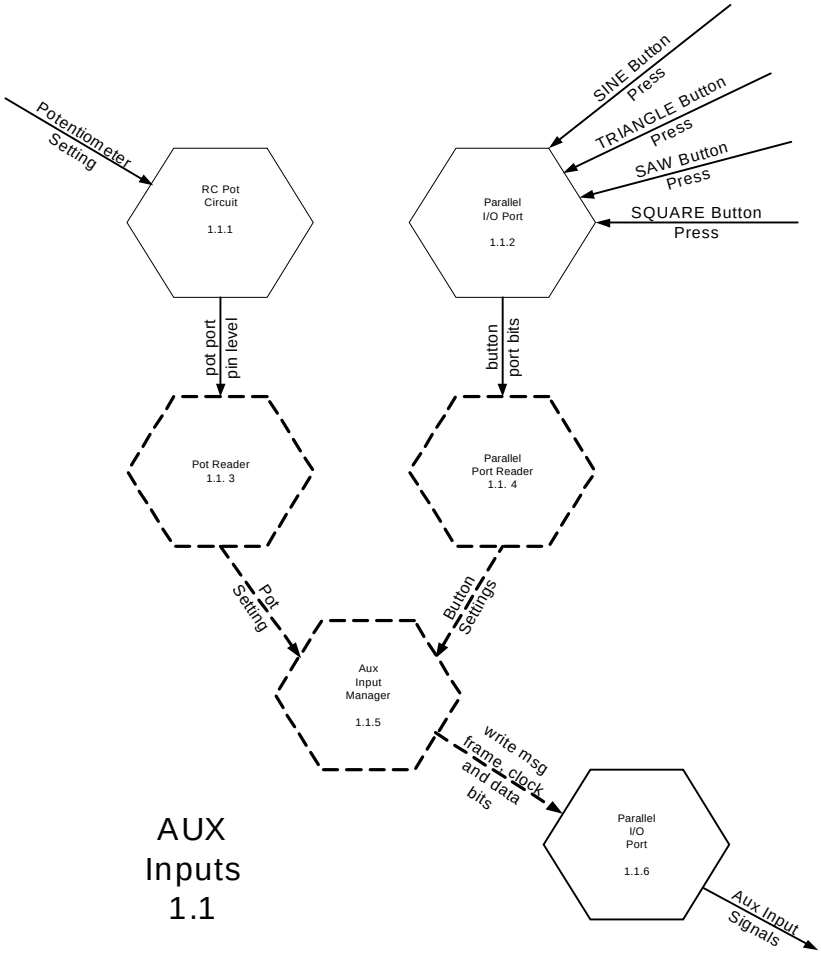
1.4 The Signal Processor

This object receives audio input data from the Input Buffer Stream, determines whether a harmonically structured voice is present, and if so, substitutes one of four synthesized voices.

1.5 Case and Circuit Boards

This hardware object includes the circuit boards for the two processors, their power supplies and associated hardware, including memory chips, buttons, knobs, LEDs, and so on. This object is assumed to supply memory, CPU cycles, and power to all the other objects that might need them.

In the Context Diagram and the Level 1 Diagram, the entire system is represented. Lower level diagrams focus on pieces of the system shown in a higher level diagram. At level 2 there will be one diagram each for Aux Inputs, Audio I/O, User Interface, and Signal Processor. Since this architecture is focusing on software, the Case and Circuit Board component will not be further explored. For the same reason, hardware-only objects will be included in the diagrams, but not supplied with text descriptions.



1.1.3 Pot Reader

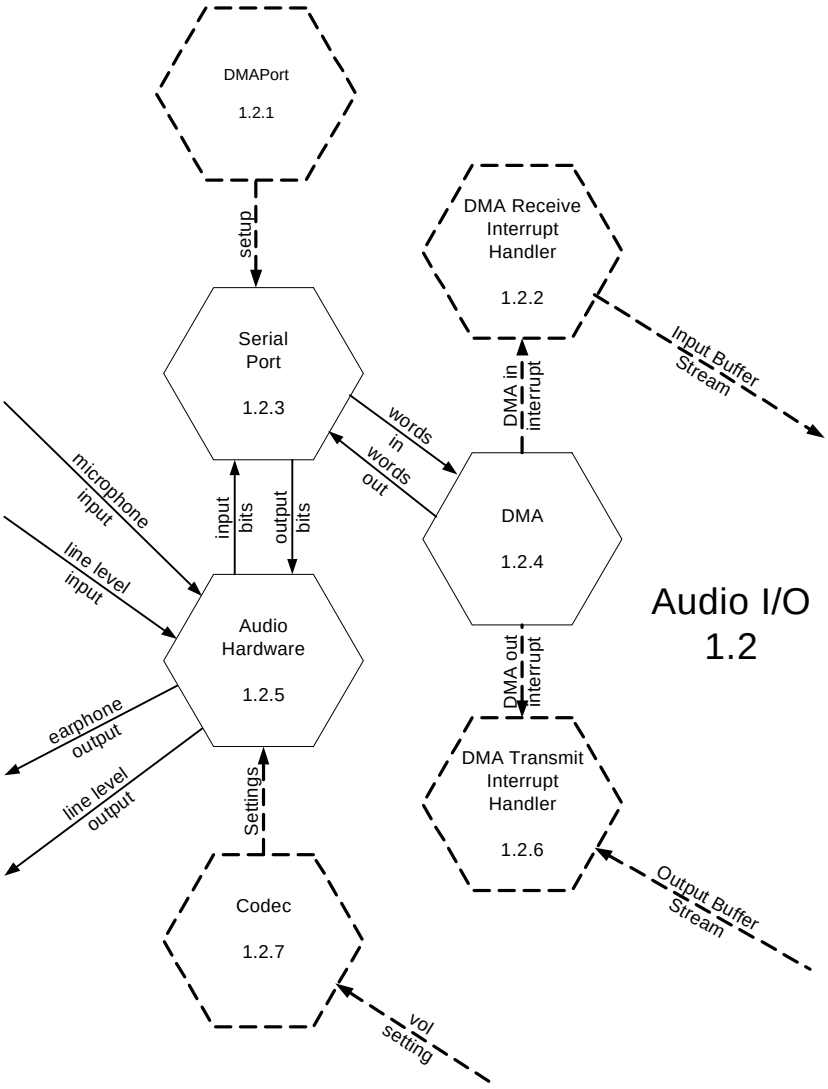
This object charges up the capacitor in the RC circuit which includes the potentiometer. When the capacitor is charged, a port pin attached to one side goes high. By timing how long it takes for the capacitor to discharge, and allow the port pin to go low, it is possible to determine the potentiometer resistance, which reveals its setting.

1.1.4 Parallel Port Reader

This object debounces the buttons, and passes their settings back to the Aux Input Manager when requested.

1.1.5 Aux Input Manager

This object has a main loop which repeatedly gets a potentiometer reading from the A/D Reader object, gets button settings from the Parallel Port reader object, and cranks the bit-banger state machine of the Synchronous Serial Port object to send messages with this information across the serial bus to the DSP board.

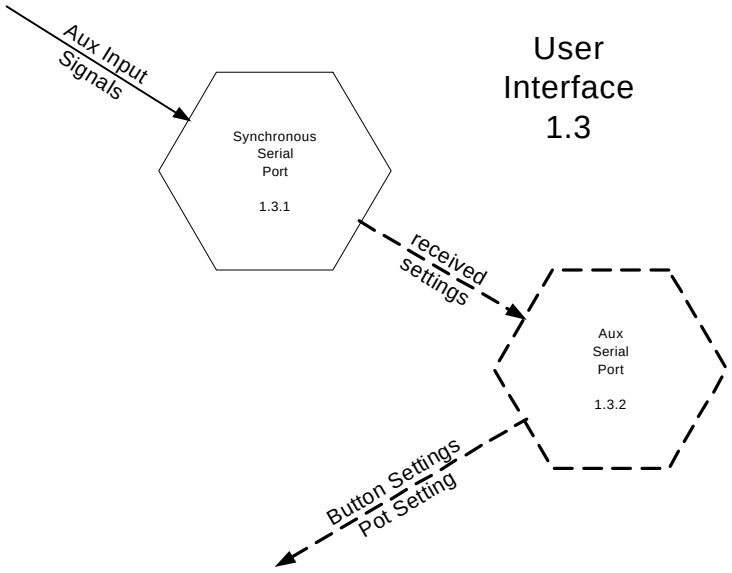


1.2.1 DMAPort
This object initializes the serial port used by the Audio I/O object.

1.2.2 DMA Receive Interrupt Handler
This object obtains buffer space in which to receive DMA input, and sends the resulting buffer to the Signal Processor object.

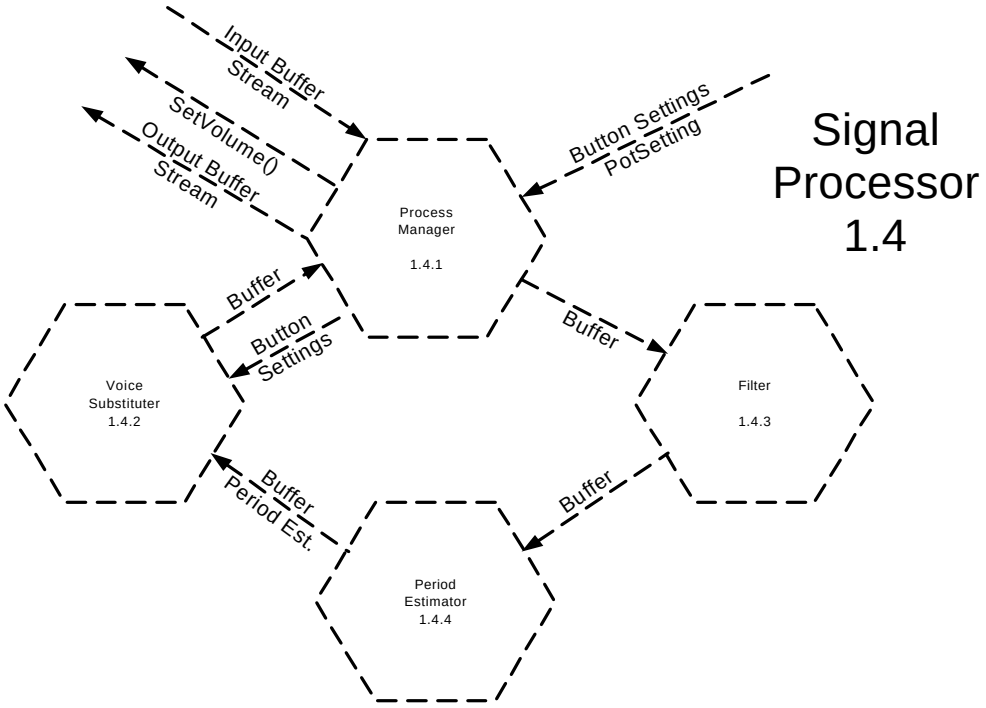
1.2.6 DMA Transmit Interrupt Handler
This one receives output buffers from the Signal Processor object, and causes the output DMA to send them back out.

1.2.7 Codec
This object initializes the Codec which handles A/D and D/A conversion. It also controls the volume of the output.



1.3.2 Aux Serial Port

This object receives the serial port interrupts, maintains a list of current button and potentiometer settings, and answers queries about those settings.



1.4.1 The Process Manager

The Process Manager receives audio input data from the Input Buffer Stream and obtains button and potentiometer settings from the User Interface. It directs the computation of data for the Output Buffer Stream, which has a synthesized voice substituted for the voice, if any, present in the input buffers.

1.4.2 The Voice Substituter

This Voice Substituter accepts a voice/novoice decision and an estimated pitch period from the Period Estimator. It receives button settings from the Process manager. If a voice is present in the input buffer, it synthesizes a voice with the requested waveform, and the same average amplitude as the input signal, and places it into the outgoing buffer. If no voice is present in the input signal, the voice substituter just passes the buffer along to the output.

1.4.3 The Filter

The Filter low pass filters the signal in the incoming buffer, and passes the filtered signal along to the Period Estimator.

1.4.4 The Period Estimator

The Period Estimator attempts to estimate the fundamental pitch frequency in the input buffer. If it cannot, it sends a no-voice decision to the Voice Substituter. If it can, it passes along the estimated pitch period instead.

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.

Time after time, 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. That's why they enjoy it.

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 phase 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

With COCOMO, you first 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, by Barry Boehm.

I wrote a script to accept the parameters and compute the formulae.
For our Voice Substitution device example, I guessed 1500 lines of code, and for parameters, picked:

mode = Organic
Analyst capability = high
Application experience = high
Complexity = nominal
Database Size = low
Language Experience = high
Modern Programming Practices = high
Programmer Capability = high
Required Reliability = nominal
Requirements Changes = nominal
Schedule Pressures = nominal
Main Storage Constraint = nominal
Execution Time Constraint = high
Use of Software Tools = nominal
Computer Turnaround Time = low
Virtual Machine Experience = nominal
Virtual Machine Volatility = nominal

The magic formulae returned the following values:

Total Man months=2.33
Total Calendar months=3.45
Developers Required = 0.68

Modified Function Point Estimating

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 collection of projections.

For the voice substitution device example, the number of function points came out to be 87. Using formulae in a script program someone sent me (thanks Kevin), the estimate came out to be:

1. Function Points = 87
2. Paper Deliverables = 170 pages, 68000 words
3. Fnct. Pts. Growth = 6.1 percent by delivery

3a. Critical Creep = 12.3 percent

4. Test Cases = 213; test runs = 850
5. Defect Potential = 266
6. Defects Shipped = 7.5
7. Schedule (months) = 6.0
8. Build Staffing = 0.58
9. Maintenance Staff = 0.17

9b. Years of Use = 3.05

10. Total Effort = 3.46 man months

The substitution of object/influence count for function points was completely arbitrary. Some of these numbers may be bogus, but some of them 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.

Add up The Guesses

In this method, you write down everything you are going to do in the project, and guess how long each action will take. Here's what I came up with for our voice substitution device.

Activity	*Hours*
Figure out why I'm doing this	1
Do use cases	1
Write detailed requirements	3
Do architecture diagrams	8
Estimate project effort	4
Research algorithms	8
Initial Class Diagram	4
Interaction Diagram for Steady State	4
Interaction Diagram for Initialization	1
Class Descriptions	24

Set up development environment	16
Write a simple straight-through program to test the hardware and development environment	8
Set up IDE editor and build	4
Code & Debug Aux Inputs classes	24
Code & Debug Audio I/O classes	80
Code & Debug User Interface classes	24
Code & Debug Signal Processor class	64
Code & Debug Streaming Buffer class	32
Integrate, test, and revise	120
Devise tests to verify key algorithms	16
Perform tests to verify key algorithms	24
Rework based on tests	40
Retest	16
Devise test to demonstrate functionality	24
Perform test to demonstrate functionality	24
Total Hours	574
Man Months	3.6
Calendar Months = Man Months*4/3	4.8

The factor of 4/3 in Calendar months came in because a quarter of my time is usually taken up with things like cleaning the office, chatting with co-workers, feeding the cat, taking phone calls, and seeking more work.

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 voice substitution device is going to take somewhere in the neighborhood of 5 calendar months to complete. It will require anywhere from a half to three quarters of my time during those 5 months.

Just so you know, the actual project took three and a half months at half time. I'm guessing that I overestimated because most of my estimating has been done for multi-person projects, which inherently take longer than single person projects, because of communication friction.

HARDWARE SUPPORT

Typically, a electrical engineer develops a printed circuit board for the product at the same time you are writing requirements and doing the architecture. Oftentimes, a mechanical engineer simultaneously develops the product case.

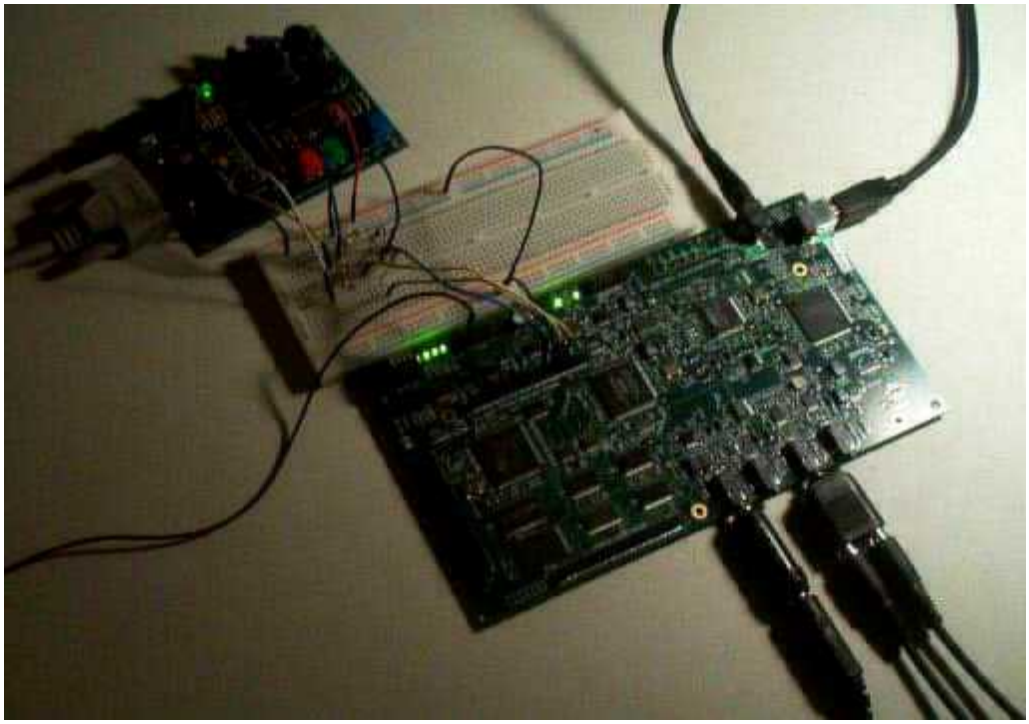
The hardware engineers can be helpful with things like getting the right power supply for the prototype boards, making sure the processor is running, providing standoffs for the circuit board to keep it off of the desktop, assembling and disassembling cases, and learning the history of the product.

In return, the other engineers 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 voice substitution device project, there are no hardware engineers. I have therefore put together a system from mostly off-the-shelf components. This system is shown in the picture below.



The main components of the system are, from top to bottom:

1) A Basic Stamp Activity Board, 27905, with a Basic Stamp2 daughter board from www.parallaxinc.com. This combination is used to provide the processor, buttons, and potentiometer for the Auxiliary Input piece of the project. This gear comes from Parallax Inc.

2) A breadboard with voltage divider networks which shift the level of the serial bus signals from 5V on the Basic Stamp board to 3.3V on the DSP board.

3) A Spectrum Digital evaluation board for the TI 5416 DSP chip. The board is called TMS320VC5416 DSK. At the time of writing this book, the board was available from Texas Instruments on their web site www.ti.com. The board comes with TI's Code Composer Studio software, and a Technical Reference manual. See the Bibliography for more information.

DESIGN

Design picks up where architecture leaves off. Its purpose is to describe a collection of virtual objects, in both the virtual machine and application layers, which work together to meet the system requirements.

Before we discuss design further, it will be helpful to review some definitions, and add a couple of new ones.

MORE DEFINITIONS

Every component of an embedded system which interacts with other components is called an **object**. It is distinct from other objects, and has an internal state, and a repertoire of behavior, which it may exhibit. 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. 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.

A **class** is a collection of zero 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.

A message to a virtual object is typically a synchronous function call to one of the methods of its class. It may also be an actual queued message, which is interpreted by the object some time after the message was queued.

You may want to consult other sources for further explanation of some of the concepts. See, for example, Chapter 3, *Classes and Objects*, in Booch's book Object Oriented Design.

METHODS OF DESIGN

At the time of this writing, an object-oriented design methodology is popular, as is a modeling language called Unified Modeling Language (UML). UML supports a large number of useful diagram types, including Class Diagrams, Interaction Diagrams, Swim Lane Diagrams, Concurrency Diagrams. Many of

the diagram types are potentially useful.

Teams often use UML modeling conventions for their projects. Some unfortunate teams purchase UML-based CASE tools and prescribe slavish conformance to nit-picking, standard modeling techniques. While there may be some justification for this practice in gigantic projects, ***Undue emphasis on the form of a design may stifle the creativity which produces its substance.***

A better approach is to select some helpful modeling tools, use whatever diagramming tool and appearance you like, and focus on the problem at hand, not the modeling convention. For our voice substitution example, we will use two modeling techniques from UML: a class overview diagram and an object interaction diagram. We will also create a class catalog.

Table 1 -- Algorithm for doing an object-oriented design	
Step 1	Create a collection of **Object Interaction** diagrams for the system. Use the interaction diagrams to explore ways to divide the functionality among the objects. Produce diagrams which show the overall operation of the system, as well as significant details. Let the architecture guide you in choosing the original objects.
Step 2	Summarize the object interaction diagrams by creating a **Class Overview** diagram. This shows each of the classes in the system, and the data and methods of each. *Repeat steps 1 and 2 until satisfied with the Object Interaction and Class overview diagrams.*
Step 3	Create a **Class Catalog** which describes each class in the system, its encapsulated data structures, its public methods, and significant private methods.

The next few pages record the outcome of the above three step process for our voice stress recorder. Then we consider briefly methods to **design in safety**.

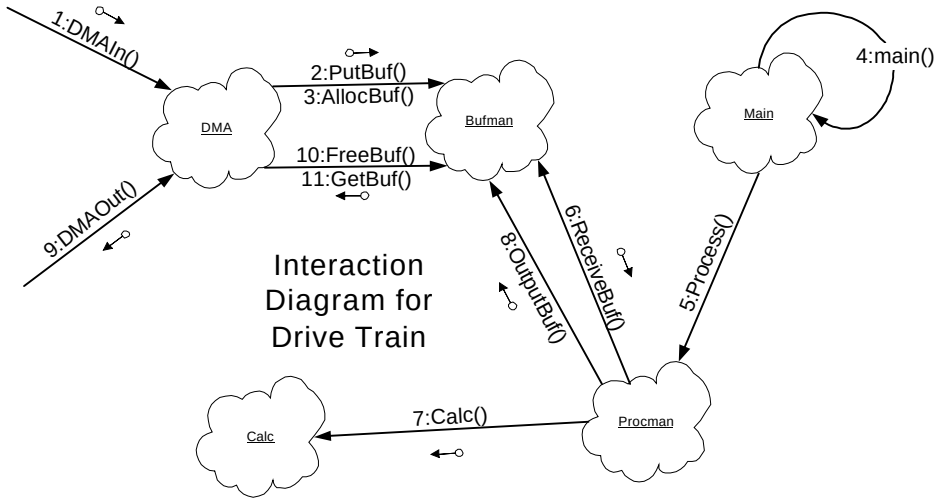
Object Interaction Diagrams

A building architect does not show a closeup of every footing and every joint in the building. Instead she shows the overall floorplan and appearance, and one of each kind of footing, joint, or railing.

A key to good design is to focus on the overview and on significant detail.

Whatever the embedded system does, it is best to start design work with its drive train. This is the object interaction path that produces the primary motivated activity of the system. It may also be necessary to produce separate

interaction diagrams for initialization, shutdown, and for individual use cases.

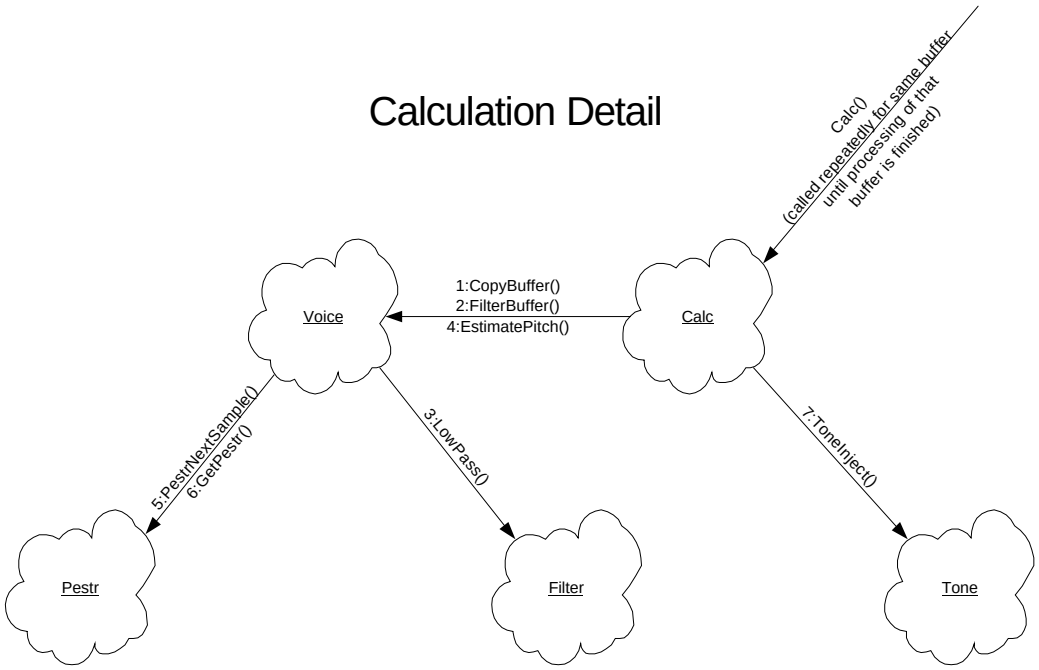


The above diagram shows the drive train in steady state operation. The numbering is the sequence in which things happen. It is useful to explain the diagram by listing the message number and describing what is going on.

Number	Message	Activity
1.	DMAIn()	The DMA has completed a complete buffer of input from the codec input serial port. It stops whatever the processor is doing, and calls this input interrupt routine.
2.	PutBuf()	DMAIn() calls PutBuf() to notify the Buffer Manager that the buffer it has been filling is ready for whomever wishes to see it.
3.	AllocBuf()	Then DMAIn() calls this buffer manager routine to obtain a pointer to a new buffer to receive DMA incoming data. It sets up the input DMA channel to begin receiving data into this new buffer.
4.	main()	There is a loop in the main program which calls the Process() method of the Procman object.
5.	Process()	If Process() has finished with the last buffer, it starts checking for the availability of a new buffer by calling Buffer Manager's ReceiveBuf() routine.
6.	ReceiveBuf()	ReceiveBuf checks for an available buffer in its process list. If there is one, it returns its address to Process(). If not, it returns NULL to Process().
7.	Calc()	Once it has a buffer, Process() calls the voice substitution device's Calc() routine, which performs all of the computation for voice and pitch detection, as well as placing the synthesized voice into the buffer for output. (See the calculation detail diagram

		for more information)
8.	OutputBuf()	When it has finished the voice substitution calculation, the Process() routine calls the Buffer Manager's OutputBuf() routine with the modified buffer. This routine queues the indicated buffer for output the next time the output DMA requests a buffer.
9.	DMAOut()	When the output DMA channel finishes the latest output buffer passed to it, it interrupts whatever else is happening and calls this output channel interrupt routine.
10.	FreeBuf()	DMAOut() calls the buffer manager's FreeBuf() routine to make the old buffer available to the next input interrupt routine.
11.	GetBuf()	Next DMAOut() calls the buffer manager's GetBuf() routine to get a pointer to the next output buffer to be sent. It sets up the DMA to use this buffer to feed the output serial port.

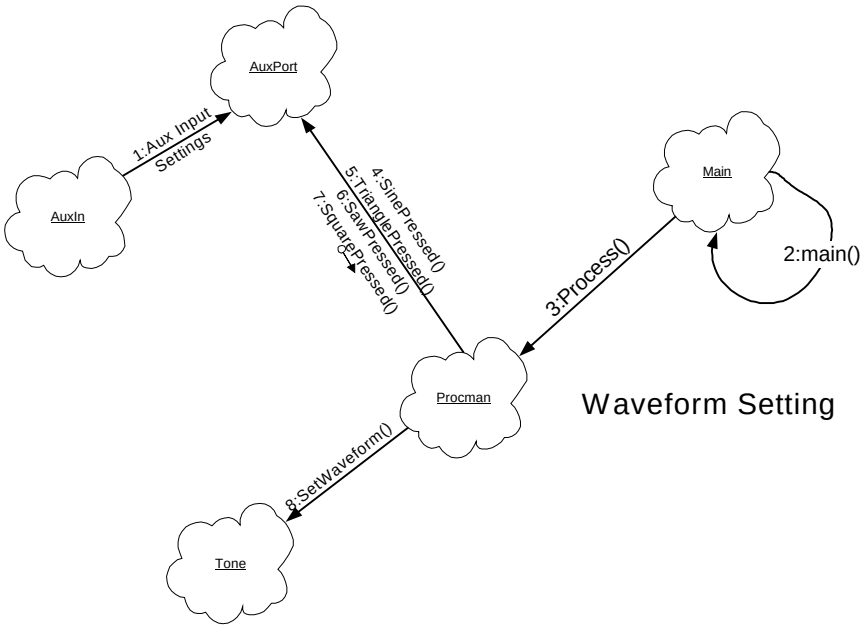
The Calc object uses four other objects, called Filter, Voice, Pestr, and Tone to accomplish its mission. The interaction diagram for the calculation is shown below:



Number	Message	Activity
	Calc()	Once it determines it has a buffer to process, the Process() method of the Procman object calls Calc() repeatedly until Calc() reports it is finished with the buffer. Each time it is called, Calc() executes the next state of the state machine responsible for performing the filtering the buffer, performing voice pitch calculation, or putting a synthesized signal into the buffer for output.
1.	CopyBuffer()	Copies all the data from the left channel of the input buffer into a special buffer in the Voice object. Also computes and saves the average absolute signal value of all the left channel samples in the buffer.
2.	FilterBuffer()	Calls the Filter object LowPass() method to filter the contents of the buffer in the Voice object.
3.	LowPass()	Performs an IIR low pass filtering operation on the contents of the buffer passed to it.
4.	EstimatePitch()	Calls PestrNextSample() routine with every sample in the Voice ob ject buffer. Then

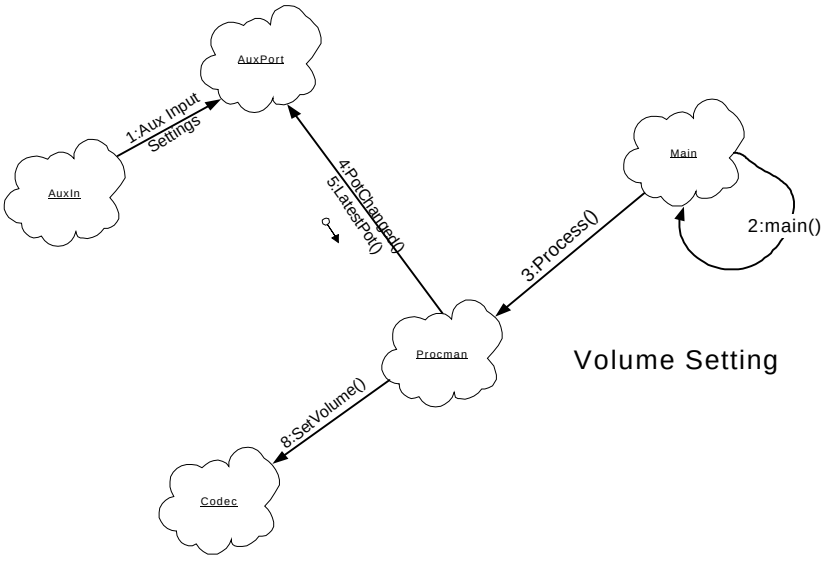
		calls GetPestr() to get the resulting pitch period estimate for the buffer.
5.	PestrNextSample()	Runs the next sample through the Pestr object's pitch estimation algorithm, updating six parallel pitch estimators, based on the heights and timing of signal peaks and valleys.
6.	GetPestr()	Looks for coincidences among the parallel pitch estimators, and returns either a pitch period estimate, or an indicator that there was no agreement among the estimators.
7	ToneInject()	If a valid pitch estimate was returned by GetPestr(), calc calls this routine to put a synthesized waveform into the output buffer.

The drive train diagram did not show the interactions involved in setting the waveform or adjusting the volume. The first one we will consider is waveform setting, which is shown below.

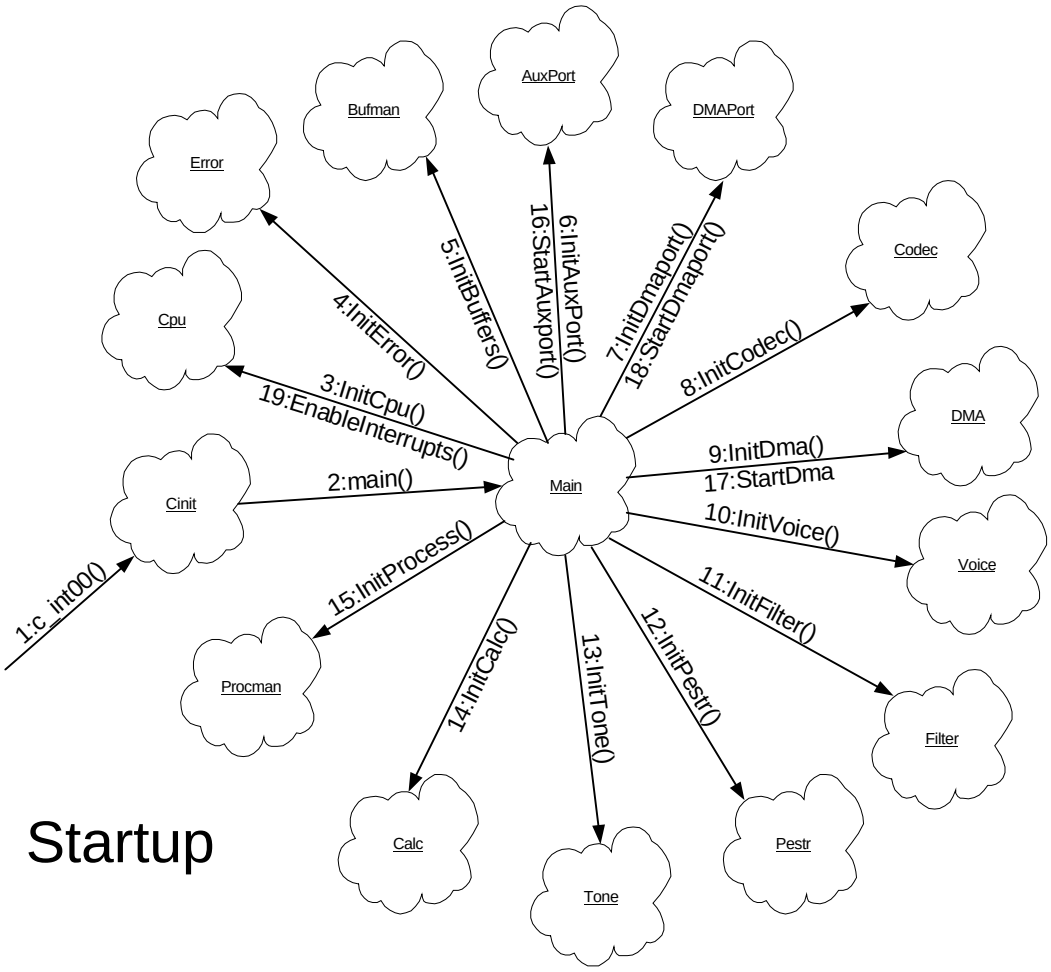


We could have included this in the drive train diagram above, but it would have disrupted the numbering scheme, and complicated the diagram.

Number	Message	Activity
1.	Aux Input Settings	The Aux Input object periodically sends the button and potentiometer settings to the Port Reader object
2.	main()	The next time main() reaches that point in its loop, it calls the Process() routine, which checks for new user inputs before processing each buffer.
3.	Process()	Procman object is responsible for the signal processing calculations. Immediately before processing a new input buffer, it checks to see if a new output waveform has been requested. If so, it calls the SetWaveform() method of the Tone object.
4,5,6,7	SinePressed() TrianglePressed() SawPressed() SquarePressed()	Each method returns TRUE if the corresponding button has been pressed.
8.	SetWaveform()	Notifies the Tone object what waveform to inject next time a voice pitch is detected.



Number	Message	Activity
1.	Aux Input Settings	The Aux Input object periodically sends the button and potentiometer settings to the Aux Port object
2.	main()	The next time main() reaches that point in its loop, it calls the Process() routine, which checks for new user inputs before processing each buffer.
3.	Process()	Procman object is responsible for the signal processing calculations. Immediately before processing a new input buffer, it checks to see if a new potentiometer setting has been received by calling PotChanged() in the Aux Port object. If one has been received, it then calls LatestPot() to obtain the value of the setting. It then converts the pot setting to a volume setting that will be understood by the codec object.
4,5	PotChanged() LatestPot()	PotChanged() returns TRUE if the potentiometer setting has been changed, FALSE if it has not. LatestPot() returns the latest potentiometer setting.
6.	SetVolume()	Sets the output channel attenuation for the codec.



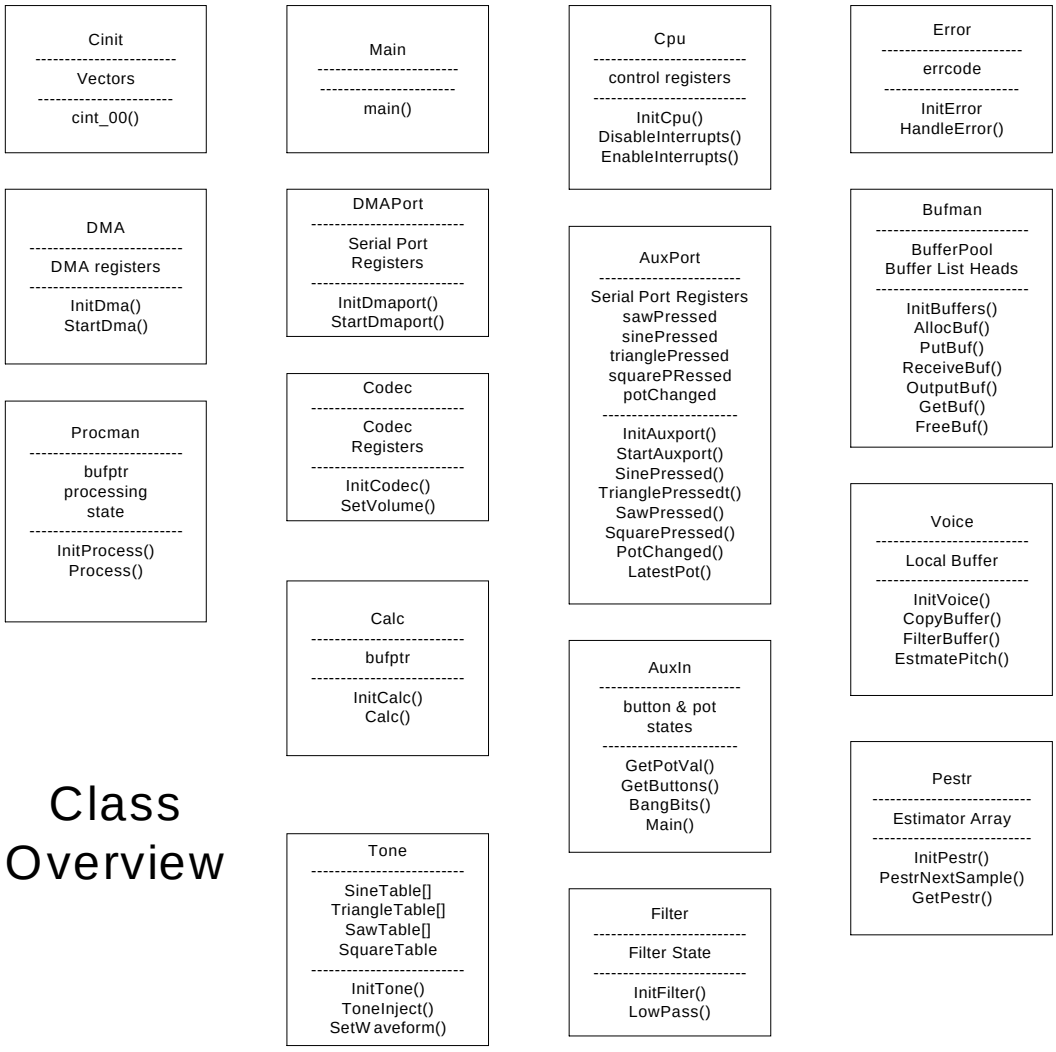
Startup

Number	Message	Activity
1.	c_int00()	Reached by the Reset vector, this method does any processor initialization that must be done immediately. Then it copies data to linker segments which require initialization, sets up the language stack, and branches to the main() method of the Main object.
2.	main()	This method carries out more initialization of the processor, and then initializes the virtual objects of the virtual machine and application layers.
3.	InitCpu()	This method sets up the phase locked loop for the CPU clock, performs any necessary initializaiton of chip selects and memory banks.
4.	InitError()	This one handles any initialization needed by the Error object. This could include handling warmstarts if they are supported, or setting up an

		error log if such is used.
5.	InitBuffers()	This routine initializes the Bufman object by placing all of its buffers onto the free buffer list.
6.	InitAuxPort()	This sets up the synchronous serial port used to accept input from the AuxInputs object. It unmasks its interrupt as well, though at this time, interrupts are still disabled.
7.	InitDMAPort()	Init the serial port which services the DMA.
8.	InitCodec()	Init the fancy A/D D/A converter.
9.	InitDMA()	Initializes the DMA object by setting up the DMA channels that it will use to input and output audio buffers via the serial port and the codec. Unmasks the DMA interrupts.
10.	InitVoice()	Sets up the voice object
11.	InitFilter()	Initializes the low pass filter object
12.	InitPestr()	Initializes the period estimator algorithm
13.	InitTone()	Initialize the tone injector tables.
14.	InitCalc()	Initializes the stress calculation routine by initializing its decimators, spectral estimators, and energy estimators. Sets default values for sensitivity and volume.
15.	InitProcess()	This initializes the Procman object, and its state machine cranked by the Process() method.
16.	StartAuxport()	Start the SPI slave port to the AuxIn object
17.	StartDmaport()	Enable DMA I/O McBsp0 port
18.	StartDma()	Enable DMA input and output channels
19.	EnableInterrups()	Once everything is set up and initialized, enabling interrupts allows the fun to begin. The main() routine enters a loop continually calling Process().

Class Overview Diagram

Whether this is the first iteration of steps 1 and 2 in the design process, or the Nth, you do the same thing. Make a overview diagram of all the classes that appear in an object interaction diagram, and any classes you know you will need, which are not in an interaction diagram. The overview diagram might look something like the one below.



UML provides modeling techniques within the class overview diagram to handle inheritance, composition of a class from a collection of other classes, the is-an-instance-of relationship, template variables, cardinality relationships between classes, and so on. It is enough to know that those features are available if you need them. See UML in a Nutshell for more information.

Class Catalog

The *Class Catalog* describes each class in the system, its encapsulated data structures, its public methods, and significant private methods. Each class corresponds to one or more source modules. The purpose of the catalog is to supply enough information for a programmer to code the modules associated with each class.

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

Class Name: <name of class>
< state the purpose of the class>
Cardinality: <Number of expected objects of this class>
Source Modules:
<names of source files for this class>
Data Elements:

Name	*Type*	*Purpose*

Public Methods:

Name	*Return Type*	*Arguments*	*Purpose*

Private Methods:

Name	*Return Type*	*Arguments*	*Purpose*

Diagrams,Algorithms, etc:
<Diagrams of any algorithms, state machnes, or flowcharts that are essential to understanding what the code must do>
Verification:
<Suggestions for verifying proper operation of this class>

Class Name: Cinit

Most embedded systems have language-dependent actions that must be dealt with at startup and shutdown. This class handles those issues. It also supplies a logical place to put the interrupt vectors.

Cardinality: 1

Source Modules: cinit.asm

Data Elements:

Name	*Type*	*Purpose*
ResetVector	interrupt vector	The processor reads this to determine where to go at powerup. It should point to the start() method of this aloha object.
DMAInputVector	interrupt vector	The processor reads this to determine where to go when the DMA input channel completes. It should point to the DMAIn() method of the DMA object.
DMAOutputVector	interrupt vector	The processor reads this to determine where to go when the DMA output channel completes. It should point to the DMAOut() method in the DMA object.
AuxPortVector	interrupt vector	The processor reads this to determine where to go when the AuxPort serial receive interrupt happens. It should point to the AuxPortRx() method in the AuxPort object.
DisableVector	interrupt vector	This is an interrupt vector that is chosen to implement the DisableInterrupt() method of the the Cpu object, by simply doing a return, rather than a return from interrupt. This bypasses any pipeline problems encountered in disabling interrupts by setting interrupt mask bits.
stack	array of words	The processor's stack pointer is set in cstart() to point to the largest address in this array.
exitcode	a word	If an exit code is supplied to the exit routine, it will be recorded here before exit() takes its final action, whatever

		that may be.

Public Methods:

Name	*Return Type*	*Arguments*	*Purpose*
c_int00()	none	none	This routine, supplied by the compiler vendor, initializes linker sections, the stack, and the heap. Then it jumps to the main program, in our case main().
exit()	none	none	This routine is called on exit from the system, normally due to an error condition.

Private Methods: none

Verification:

1. Make sure the stack location is what you expect it to be, and that its size is what you need. Normally, you can do this by looking at the link map. If the stack is not big enough or too big, or in the wrong place, modify the code in c_int00().

2. Make sure that after execution of start(), that all initialized structures contain the correct values, and that all const parameters contain their values.

3. Make sure that any code which is supposed to be in RAM is actually in RAM. If your system runs, it probably is, but verify using the link map, and a breakpoint somewhere in main().

4. If exit() is to be used by the system, make sure that the exit code is recorded whenever someone calls exit.

5. Test all the interrupt vectors to make sure they actually transfer control to where they are supposed to.

Class Name: AuxIn
This is the class whose code runs on the Basic Stamp2 processor. It reads the potentiometer and buttons, and bit-bangs words across a serial bus to the AuxPort on the DSP board.

Cardinality: 1

Source Modules:
auxin.bas

Data Elements:

Name	*Type*	*Purpose*
IOState	word	This is the state of the bit-banger state machine.
prevSine	word	Previous SINE button setting
latestSine	word	Latest SINE button setting
sineCount	word	Consecutive times SINE button has been set this way (stop incrementing at some small maximum value)
officialSine	word	Official SINE button setting (when this changes from UP to DOWN, a SINE botton pressed indication is sent in the next message to the DSP board)
prevTriangle	word	Previous TRIANGLE button setting
latestTriangle	word	Latest TRIANGLE button setting
triangleCount	word	Consecutive times TRIANGLE button has been set this way (stop incrementing at some small maximum value)
officialTriangle	word	Official TRIANGLE button setting (when this changes from UP to DOWN, a TRIANGLE botton pressed indication is sent in the next message to the DSP board)
prevSaw	word	Previous SAW button setting
latestSaw	word	Latest SAW button setting
sawCount	word	Consecutive times SAW button has been set this way (stop incrementing at some small maximum value)
officialSaw	word	Official TRIANGLE button setting (when this changes from UP to DOWN, a TRIANGLE botton pressed indication is sent in the next message to the DSP

		board)
prevSquare	word	Previous SQUARE button setting
latestSquare	word	Latest SQUARE button setting
squareCount	word	Consecutive times SQUARE button has been set this way (stop incrementing at some small maximum value)
officialSquare	word	Official SQUARE button setting (when this changes from UP to DOWN, a SQUARE botton pressed indication is sent in the next message to the DSP board)
prevPot	word	Previous Potentiometer Setting
latestPot	word	Latest Potentiometer Setting
potCount	word	Consecutive times pot setting has had this value (stop incrementing at some small maximum value)
officialPot	word	Official Potentiometer Setting (This is what is sent with each message to the DSP board)

Public Methods:

Name	*Return Type*	*Arguments*	*Purpose*
main()	none	none	This is the main loop of the program. It repeatedly calls the GetPotVal(), Debounce(), GetButtons(), wait() and BangBits().
GetPotVal()	word	none	Returns the latest potentiometer reading which has persisted for at least one cycle of reads.
Debounce()	word	none	Checks to see if new button settings and pot value are different from previous. If not, increments the count. When the count reaches its maximum value, sets the official values.
BangBits()	none	none	This is a state machine. Every time it is called, it executes one of the states which sends a three wire serial message to the DSP board. Both the state machine and message format are shown in the section on Diagrams and Algorithms. It

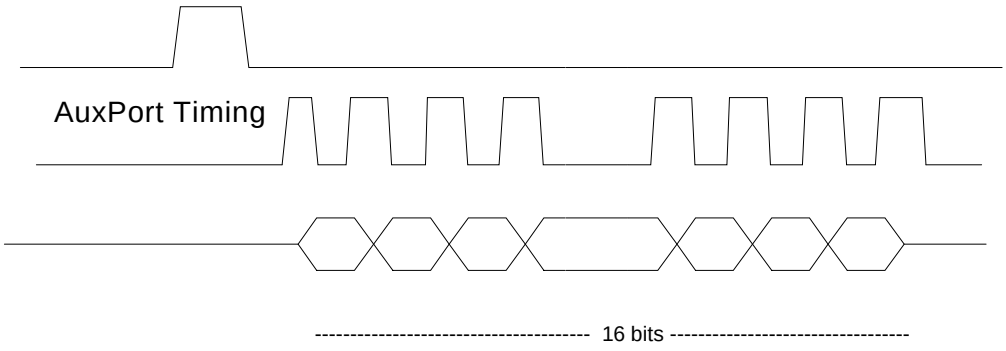
			gets its data from the official button states and official pot value.

Private Methods:

Name	*Return Type*	*Arguments*	*Purpose*
wait()	none	word	This routine waits for a period of time that varies exponentially with its single argument.

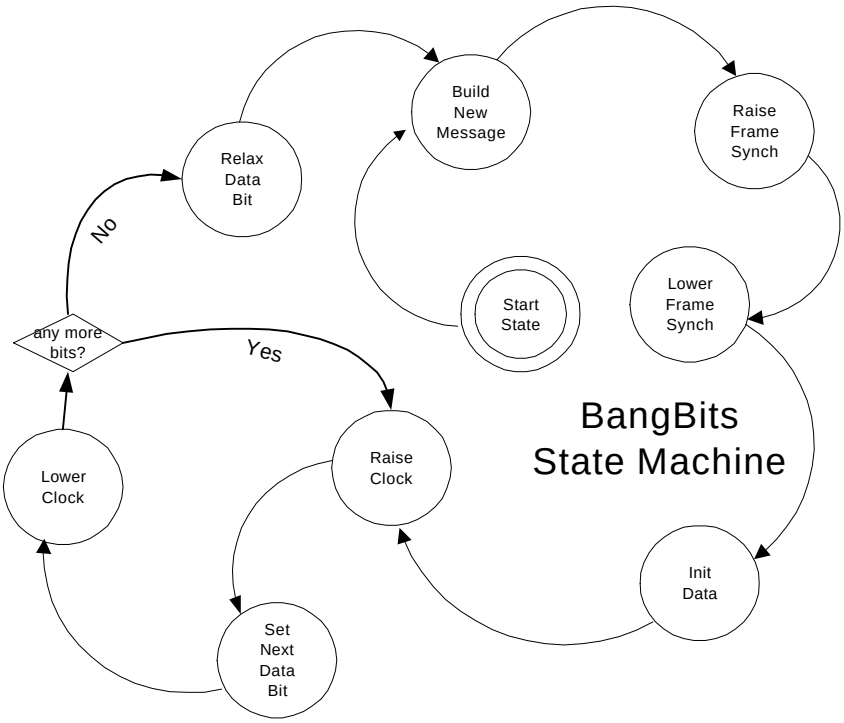
Diagrams,Algorithms, etc:

The AuxPort serial bus timing should be as shown below:



The first signal is frame synch pulse.
The second signal is the bit clock.
The third signal is the data signal. Transfer on rising edge of bit clock.
The first bit of data is the most significant. The sixteenth is the least.
The first four bits of data are the SINE, TRIANGLE, SAW, and SQUARE button statuses respectively.
The last twelve bits are the potentiometer setting.

The BangBits() state machine is shown below:



Verification:

1) Look at the serial output of the Basic Stamp2 processor on the oscilloscope. Make sure that the picture is as shown in the timing diagram. Verify that pushing the buttons makes the appropriate change in the first four bits, and that setting the potentiometer appropriately changes the next twelve bits.h

2) Verify that the Aux Serial port on the DSP board can receive the serial message put out by the AuxIn object.

Class Name: Auxport

The object of this class initializes the serial port devoted to auxilliary input, and it receives data words from the AuxIn object.

Cardinality: 1

Source Modules:

auxport.c auxport.h

Data Elements:

Name	*Type*	*Purpose*
sinePressed	bool	If TRUE, the sine button has been pressed but not yet queried via the SinePressed() method.
trianglePressed	bool	If TRUE, the triangle button has been pressed, but not yet queried via the TrianglePressed() method
sawPressed	bool	If TRUE, the saw button has been pressed, but not yet queried via the SawPressed() method
squarePressed	bool	if TRUE, the square button has been pressed, but not yet queried via the SquarePressed() method
potChanged	bool	if TRUE, the value of the pot setting has changed, but has not yet been queried by the LatestPot() method.
latestPot	word	Latest value of the Pot setting received from the AuxIn object.

Public Methods:

Name	*Return Type*	*Arguments*	*Purpose*
InitAuxport()	none	none	Set up McBSP0 to be used as an SPI slave to the Auxin object on the BasicStamp2 board.
StartAuxport()	none	none	Start receiving messages from the BasicStamp2 board.
SinePressed()	word	none	Returns the contents of sinePressed. Resets sinePressed to FALSE
TrianglePressed()	bool	none	Returns the contents of trianglePressed. Resets trianglePressed to FALSE.

SawPressed()	bool	none	Returns the contents of sawPressed. Resets sawPressed to FALSE.
SquarePressed()	bool	none	Returns the contents of squarePressed. Resets squarePressed to FALSE.
PotChanged()	bool	none	Returns the contents of potPressed. Resets potPressed to FALSE.
LatestPot()	word	none	Returns the value of latestPot.

Private Methods:

Name	*Return Type*	*Arguments*	*Purpose*
AuxportRxInterrupt()	none	none	reads McBsp0 receive register. sets sinePressed, trigangle Pressed, sawPressed, squarePressed, potChanged, and latestPot.

Verification:

1. Use the emulator to make sure that when a sine, triangle, saw, or square button is pressed, the AuxportRxInterrupt() routine is called, and it sets sinePressed, trianglePressed, sawPressed, or squarePressed TRUE.

2. Use the emulator to make sure that when the potentiometer wheel is moved, the latestPot value is changed.

3. Step through the SinePressed(), TrianglePressed(), SawPressed(), SquarePressed(), and PotChanged() routines to verify that they set sinePressed, trianglePressed, sawPressed, squarePressed, and potChanged to FALSE.

Class Name: Bufman

This class provides the buffer management needed to support movement of audio data from the input to the output.

Cardinality: 1

Source Modules: bufman.h bufman.c

Data Elements:

Name	*Type*	*Purpose*
BUF	a data structure definition	A BUF is a structure which defines a single data buffer. It contains the fields defined below:
bufcount	32 bit integer	BUF field: buffer sequency number set on allocation
nextBuf	bufArray index	BUF Field: index of next buffer in any chain this buffer happens to be linked into.
data	16 bit integer array[480]	BUF Field: This is where all the data for the left channel of the incoming codec data stream goes. It is 16 bits per sample, 48000 samples/second. Each sample is padded by anothe 16 bit data word. This means that the data array must hold 480 sixteen bit words.
buffers	an array of BUF structures[5]	This array contains all of the buffers used by the application
totCount	32 bit integer	incremented by AllocBuf() everytime it allocates a new buffer.
firstFree	buffers index	Index of the first buffer in the free list.
inBuf	buffers index	Index of a buffer that has been alloc'd but not yet put. It is being used for DMA input.
firstProcess	buffers index	Index of the first buffer in the process list. The process list is a FIFO list.
calcBuf	buffers index	index of buffer on which calculations are performed.
firstOut	buffers index	Index of the first buffer in the output list. The output list is a FIFO list.
outBuf	buffers index	Index of a buffer that has been gotten, but not yet freed. It is being used for DMA output.

Public Methods:

Name	*Return Type*	*Arguments*	*Purpose*
InitBuffers()	none	none	Initialize the data structures used by Bufman. Link all buffers in bufArray onto free list
AllocBuf()	pointer to BUF or NULL	none	If the inBuf variable is not NIL, call the HandleError() routine. If there are any buffer's on the free list, return a pointer to the first one. Point firstFree at the BUF indicated by the nextFree field of the buffer returned. Set the inBuf variable to the index of the buffer whose pointer was returned. Increment totCount. If there is no buffer on the free chain, return NULL.
PutBuf()	none	none	If the inBuf variable is NIL, call the HandleError() routine. Otherwise, add the buffer indicated by the inBuf variable to the end of the process list. Set the value of inBuf variable to NIL.
ReceiveBuf()	pointer to BUF or NULL	none	If there is a buffer in the process list, return the first buffer from the process list, and return a pointer to it. If there is no buffer in the process list, return NULL.
OutputBuf()	none	pointer to BUFF	Add the buffer indicated to the end of the output list.
GetBuf()	pointer to BUF or NULL	none	If the outBuf variable is not NIL, call the ErrorHandler(). If there is a buffer available on the output list, remove the first buffer from the output list, and put it's index into the outBuf variable.

FreeBuf()	none	none	If the outBuf variable is NIL, call the HandleError() routine. Otherwise, add the buffer indicated in the outBuf variable to the end of the free list. Set the value of outBuf variable to NIL

Verification:

1. Make sure that buffers added to the process list with PutBuf() are always received by ReceiveBuf() in the same order that they were put. Make sure that buffers added to the output list by OutputBuf() are always received by the caller of GetBuf() in the same order in which they were added.

2. Verify that during normal operation, the totalcount read from each buffer increases monotonicaly.

3. Verify by adding a temporary delay to the buffer processing, that if more than two buffers are used, that they continue to arrive at the output in the correct order, as shown by the totalcount field of the buffers.

4. Stop at a breakpoint after the system has been running for a while, and verify that you can account for the whereabouts of all your buffers. Do they all appear in either the free list or one of the processing lists, including inBuf, firstProcess, calcBuf, firstOut and outBuf?

Class Name: Calc
This class does the necessary signal processing on each buffer of audio data as it is presented by the process manager Procman.

Cardinality: 1

Source Modules:
calc.c calc.h

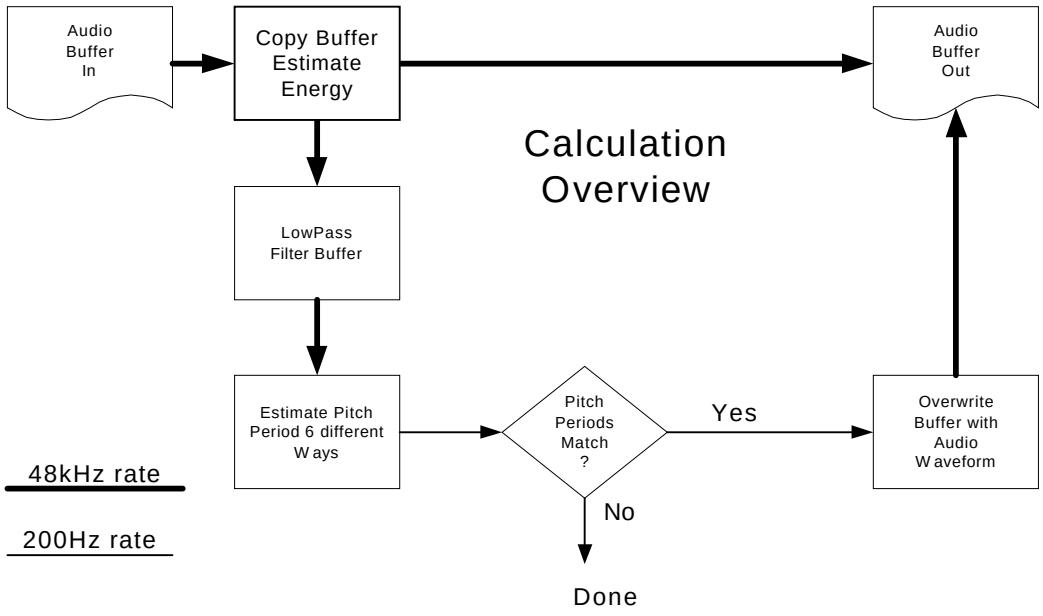
Data Elements:

Name	*Type*	*Purpose*
cstate	word	Calc() state. Initialized to 0.
bufptr	word ptr	points to buffer on which calculation is to be performed.
avgAmplitude	word	average absolute amplitude of the buffer being worked on.
pitchFreq	word	pitch frequency for the buffer being worked on

Public Methods:

Name	*Return Type*	*Arguments*	*Purpose*
InitCalc()	none	none	Initialize the cstate and buffer pointer
Calc()	word	BUF pointer	Executes the algorithm shown in the Calculation Overview shown in the diagrams section. NOTE: The Calc() algorithm is divided up into chiunks, or states, which are executed on successive calls to Calc() with the same buffer. Each time Calc() finishes with a chunk of computation, it returns a positive integer. When it finishes the last chunk, it returns 0. The number Calc() returns is the state of the calculation. Before it is called the first time with a new buffer, the state is 0.

Diagrams,Algorithms, etc:



The diagram above gives an overview of the algorithm carried out in response to the Calc() method, which is executed in its entirety 200 times/second. Each of the steps shown is discussed in more detail below.

Copy Buffer Estimate Energy -- Each sample the incoming audio buffer is copied to a local buffer for further analysis, and an estimate is made of the average absolute value of the signal.

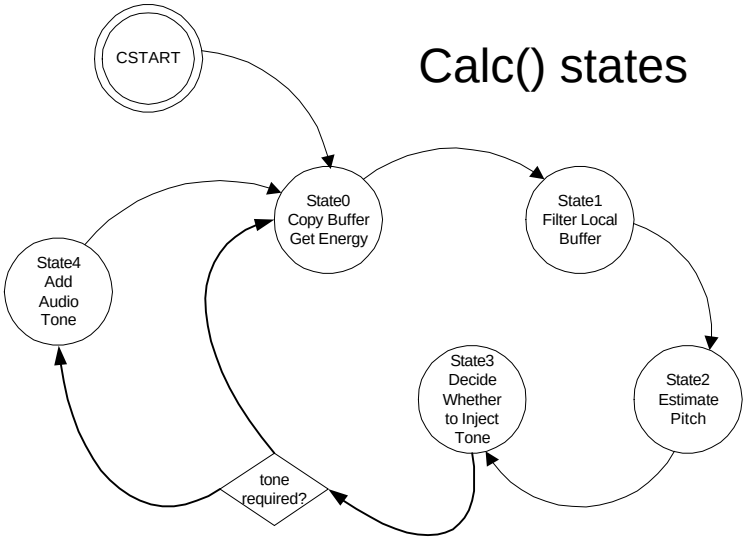
Filter Buffer -- Low pass filter the local buffer to remove energy over 1 kHz.

Estimate Pitch 6 Different Ways -- Six different pitch estimators are computed based upon the values of signal peaks and valleys and the times between them. The algorithm used is from Rabiner and Gold, pp681-687.

Pitch Periods Match -- A matching algorithm is used to determine how many of the pitch estimates match each other, and a previous history of pitch estimates. If enough of the current and historical periods match, then the decision is reached to declare the buffer "voiced" with a particular fundamental frequency.

Overwrite Buffer with Audio Waveform -- Copy samples from the current waveform table into the output Audio buffer, setting the amplitude as described in the energy calculation, and the frequency as described by the pitch frequency.

The complete state machine for Calc() is shown below:



Alright, I admit that there is no need, based upon the current requirements, to break the calculation up into separate states, each executed by a single call of Calc() from the Procman object. It's just a habit, which may prove helpful if new features are added which need to be serviced concurrently by the background loop.

Verification:

1) Verify that the potentiometer setting varies the volume from silence to a usable volume in the earphones, or a speaker if one is attached.

2) Verify that input of a voiced audio signal results in output of a voiced output signal with the waveform selected by one of the waveform buttons.

3) Verify that sweeping the input voice frequency smoothly from 50Hz to 880 Hz sweeps the output frequency smoothly over the same range.

4) Verify with speech input that the voiced portions of the speech are reproduced with the waveform changed, but that the unvoiced portions of speech are merely copied as-is to the output.

5) Verify that there is at most a single buffer period (1/200th second) delay between input and output signals.

Class Name: Codec
The single object of this class initializes the A/D D/A converter used for audio input and output. The codec used in this project is the Burr Brown codec, PCM3002. Data Sheet reference is in bibliography.

Cardinality: 1

Source Modules:
codec.c codec.h

Public Methods:

Name	*Return Type*	*Arguments*	*Purpose*
InitCodec()	none	none	Sets up the codec for use as the audio A/D D/A gadget with a sample rate of 48kHz.
SetVolume()	none	s16 range (0-255)	Sets the output attenuation for the codec. a value of 0 results in silence at the ouptut, and a value of 0xFF results in 0dB attenuation.

Verification:
1) Use a function generator to generate square waves, ramps, and sine waves, and verify that on input, the data in the input buffer has the same form as that generated.

2 For each waveform selected by a waveform button, use an oscilloscope to verify that that is the wave shape that comes out of the the codec.

Class Name: Cpu
The single object of this class is responsible for setting the interrupt vector location at powerup, setting the 5416 bankswitch registers, and setting up the clock for 160MHz operation. It also provides routines to disable and restore interrupt processing.

Cardinality: 1

Source Modules:
cpu.c cpu.h

Public Methods:

Name	*Return Type*	*Arguments*	*Purpose*
InitCpu()	none	none	Sets up the 5416 PMST register which defines, among other things the interrupt vector locations, and sets up the bank switch registers. Also sets up the phase locked loop to run the system clock at 160MHz.
DisableInterrupts()	word	none	Disables interrupts by triggering a DisableInterrupt interrupt. Returns 1 if interrupts were disabled prior to the call to this routine. Returns 0 if they were not.
EnableInterrupts()	none	none	Unconditionally enables interrupts by resetting the interrupt mask bit.

Verification**:**
1) Run a test program to verify that interrupts never happen while they are supposed to be disabled.

2) Look at the system clock with an oscilloscope, and verify that it is 160MHz.

.Class Name: DMA
The single object of this class sets up and controls the input and output DMA channels which are connected to the serial port used with the DMA.

Cardinality: 1

Source Modules:
dma.c dma.h

Public Methods:

Name	*Return Type*	*Arguments*	*Purpose*
InitDma()	none	none	Set up the input and output DMA channels to copy 960 words of data from/to the serial port used by the DMA, to a buffer area to be specified.
StartDma()	none	none	Set up some buffers for the input and output DMA channels to use, and start up those DMA channels.

Private Methods:

Name	*Return Type*	*Arguments*	*Purpose*
DMAInInterrupt()	none	none	When the input DMA channel finishes 960 words, it causes an interrupt which calls this routine. This routine calls the Bufman routine PutBuf() to notify it that the buffer has been filled. Then it calls AllocBuf() to get a pointer to an unused buffer. It sets up the DMA input channel to do the next input to the new buffer.
DMAOutInterrupt()	none	none	When the output DMA channel finishes 960 words, it causes an interrupt which calls this routine. This routine calls the Bufman routine FreeBuf() to return the output buffer to Bufman. Then it calls the GetBuf() routine to get a pointer to the next output buffer to be sent out. It sets up

			the DMA output channel to do the next output from the new buffer.

Diagrams,Algorithms, etc:

NOTES:
The time between samples is 1/48000 or 20.83333 microseconds. This means that the DMA interrupts have to be serviced and readied for the next sample within that time period. There is a remote possibility that the input and output DMA interrupts could happen at the same time, so only 10 microseconds is available to handle each interrupt.

If interrupt latency is kept to 5 microseconds, that allows, worst case, only 5 microseconds to execute input or output interrupt handling code. If the processor is running at 160MHz, we have at least 800 instruction times to handle each interrupt. This should be enough time, but if it is not, we could use a double buffering scheme for each channel, in which the DMA device continually moves data to/from a circular buffer of twice the size of the data buffer. This would allow us a fiftieth of a second to process the DMA interrupt.

Verification:

1) Measure the maximum interrupt latency in the system. Use the measured value to recompute the time available to each DMA interrupt. Verify that a DMA interrupt, including its calls to Bufman, always executes in less than the time available.

2) See tests 1) and 2) for the Codec object.

Class Name: DMAPort
The single object of this class is responsible for initializaing the serial port used to pass data between the DMA and Codec.

Cardinality: 1

Source Modules:
dmaport.c dmaport.h

Public Methods:

Name	*Return Type*	*Arguments*	*Purpose*
InitDmaport()	none	none	Set up the serial port, McBSP0, used to interface the Codec and DMA.
StartDmaport()	none	none	Enable the input and output channels of McBSP0.

Diagrams,Algorithms, etc:
We want sixteen bit transfers, framed by a signal provided by the Codec. Also, we want to transfer only left channel data to/from the Buffers, so we must trigger the serial port only on the downgoing edge of the Codec's left/right clock. The right channel data going into the buffers, or out the Codec will all be garbage.

Verification:
1) Hook a scope up to the Codec's left/right clock, bit clock, data in and data out signal lines. Make sure the signals look as they should according to the datasheets, and the nature of the data being transferred.

Also perform tests 1) and 2) in the Codec object.

Class Name: Error

The Error object provides a way to gracefully stop processing with an error code should something go very wrong. This is useful during debug for catching conditions that shouldn't happen. Hopefully, such conditions will never happen during normal operation. If they do, the Error object will stop processing in such a way that it is clear something unexpected has happened.

Cardinality: 1

Source Modules:
error.c error.h

Data Elements:

Name	*Type*	*Purpose*
errcode	word	numeric value that is keyed to the particular type of error that took place. The mapping between numeric value and error description is in error.h.

Public Methods:

Name	*Return Type*	*Arguments*	*Purpose*
InitError()	none	none	Clear errcode
HandleError()	none	word	Set errcode to the word provided. Enter an infinite loop blinking out the error code on an LED on the DSP board.

Verification:
1) Seed the system with errors of various types, and verify that the system stops with the appropriate error code in the HandleError() routine.

Class Name: Filter
This object low pass filters the local buffer owned by the Procman object, and passed to the Calc object. This is to prevent a lot of high frequency energy from interfering with peak detection in the pestr object.

Cardinality: 1

Source Modules:
filter.c filter.h

Data Elements:

Name	*Type*	*Purpose*
g	FLOATING	gain of each stage of the butterworth filter
1st biquad state	FLOATING variables	state variables for first biquad of IIR low pass filter
2d biquad state	FLOATING variables	state variables of second biquad of IIR low pass filter

Public Methods:

Name	*Return Type*	*Arguments*	*Purpose*
InitFilter()	none	none	Initialize the filter by clearing the state variables and setting the single stage gain
LowPass()	none	s16 *ptr	Apply the filter to every sample in the buffer pointed to by the argument.

Verification:

1. Modify the LowPass() routine to do no filtering whatsoever. What effect does this have on the operation of the device. There seem to be a couple of possibilities: either the computed voices are noiser representations of the incoming voices, or more time is consumed in computing high frequency peaks. The latter effect could cause the device to fail to meet its output deadline and crash.

Class Name: Main
This object's main() method receives control after the c_int00() method of the Aloha object has done its job. It initializes the virtual machine and application objects, and is the root of background processing.

Cardinality: 1

Source Modules:
main.c

Public Methods:

Name	*Return Type*	*Arguments*	*Purpose*
main()	int	none	main() performs virtual machine and application initialization, and then loops continuously calling Process().

Verification:
1) This object will be thouroughly tested when debugging the rest of the system is complete.

Class Name: Pestr

This routine maintains six time-based estimates of pitch frequency, and returns a voiced/not voiced decision and pitch period estimate when queried. It is based upon an algorithm published in Theory and Application of Digital Signal Processing by Rabiner and Gold, pp681-687 (see Bibliography).

Cardinality: 1

Source Modules:

pestr.c pestr.h

Data Elements:

Name	*Type*	*Purpose*
PESTR	struct	typedef'd structure used to calculate a single period estimate
scount	u16	sample counter of peak or valley associated with last accepted entry for this estimator.
value	s16	value of last accepted entry for this estimator (peak value, valley value, or difference between peak and previous peak, peak and previous valley, valley and previous peak, or valley and previous valley)
sample data		
currScount	unsigned int	current sample count, updated each time a new sample is passed to this object
currValue	int	value of current sample
currDelta	int	difference between current sample and previous sample
prevValue	int	value of previous sample
prevDelta	int	previous value of currDelta
peak data		
peakScount	unsigned int	sample count of latest peak
peakValue	int	value of latest peak
prevPeakScount	unsigned int	sample count of previous peak
prevPeakValue	int	value of previous peak
valley data		
valleyScount	unsigned int	sample count of latest valley
valleyValue	int	value of latest valley

prevValleyScount	unsigned int	sample count of previous valley
prevValleyValue	int	value of previous valley
pest	PESTR [6]	Array of six PESTR structures used to maintain pitch period estimates.
hpest	unsigned int[6][6]	Historical matrix of pitch period estimates used for matching. First row is latest computed pitch estimates of the six estimators. Second row is previously computed estimates of the six estimators. Third row is the six extimates computed before the previous six. Fourth row is the sum of the 1st and second rows. Fifth row is the sum of the second and third rows. Sixth row is the sum of the first three rows.
match threshold	unsigned int	This is the number of matches required between a current pitch period estimate and historical estimates before a new pitch period estimate may be returned.

Public Methods:

Name	*Return Type*	*Arguments*	*Purpose*
InitPestr()	none	none	Initialize the sample data, peak and valley data and the pitch period estimators.
PestrNextSample()	none	s16	Accepts the value of the next sample from the input buffer. Checks to see if it is a peak or valley. If it is a peak, rolls the peak data and then updates the three peak estimators. If it is a valley, rolls the valley data and then updates the three valley estimators.
GetPestr()	s16	none	This routine is called once for each input buffer. It updates the historical pitch estimate matrix, hpest. Then it searches for the most matches between current estiamtes and historical estimates. If the number of

			matches exceeds a match threshold, it returns the value of the estimate that matches the greatest number of historical estimates. If there is no match, it returns NIL, indicating that the buffer did not contain a voiced signal.

Diagrams,Algorithms, etc:

There are diagrammatic representations of this algorithm in the book by Rabiner and Gold, as well as in their article in the Journal of the Acoustic Society of America (Bibliography under Gold and Rabiner).

Verification:

1. Run the algorithm with sine wave, square wave, saw, and triangle inputs in the frequency range of 50-880 Hz. Verify that it correctly identifies the pitch frequency in all cases.

2. Run the algorithm with voice input from a microphone. Gradually sweep the voice from low range to high range and back and verify that the algorithm tracks the pitch changes.

3. Using spoken input, verify that a no-pitch decision is reached for hissing. Use a white noise source to verify that a no-pitch decision is reached for white noise input.

Class Name: Procman
The single object of this class coordinates the processing of audio buffers with the setting of the processing parameters of threshold and volume.

Cardinality: 1

Source Modules:
procman.c procman.h

Data Elements:

Name	*Type*	*Purpose*
pstate	word	current state of the process state machine.
currentBuf	BUF ptr	Points the the current buffer being processed.

Public Methods:

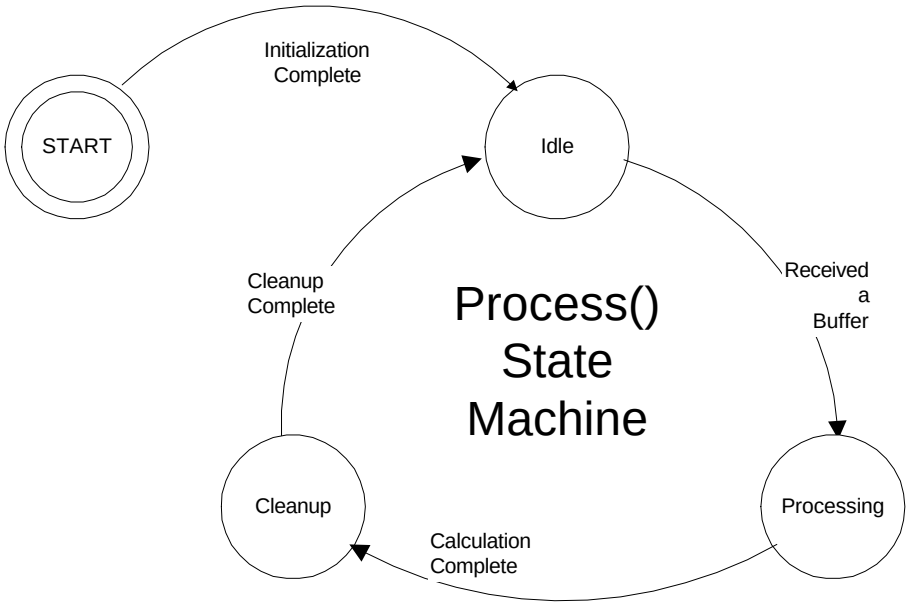
Name	*Return Type*	*Arguments*	*Purpose*
InitProcess()	none	none	Initializes the Procman object
Process()	none	none	Executes the next state of the Process() state machine. See the state machine diagram and flowcharts below.

Private Methods:

Name	*Return Type*	*Arguments*	*Purpose*
idleState()	none	none	Execute idle state logic for Pocess state machine
processingState()	none	none	Execute processing state for Process state machine
cleanupState()	none	none	Execute cleanup state for Process state machine
potToVol()	u16	u16	Convert a potentiometer setting to a coded volume setting
handleInputs()	void	void	If there are any new button or potentiometer inputs, set the volume or set the waveform as appropriate.

Diagrams,Algorithms, etc:

The state machine executed by repeated calls to Process() is shown below:



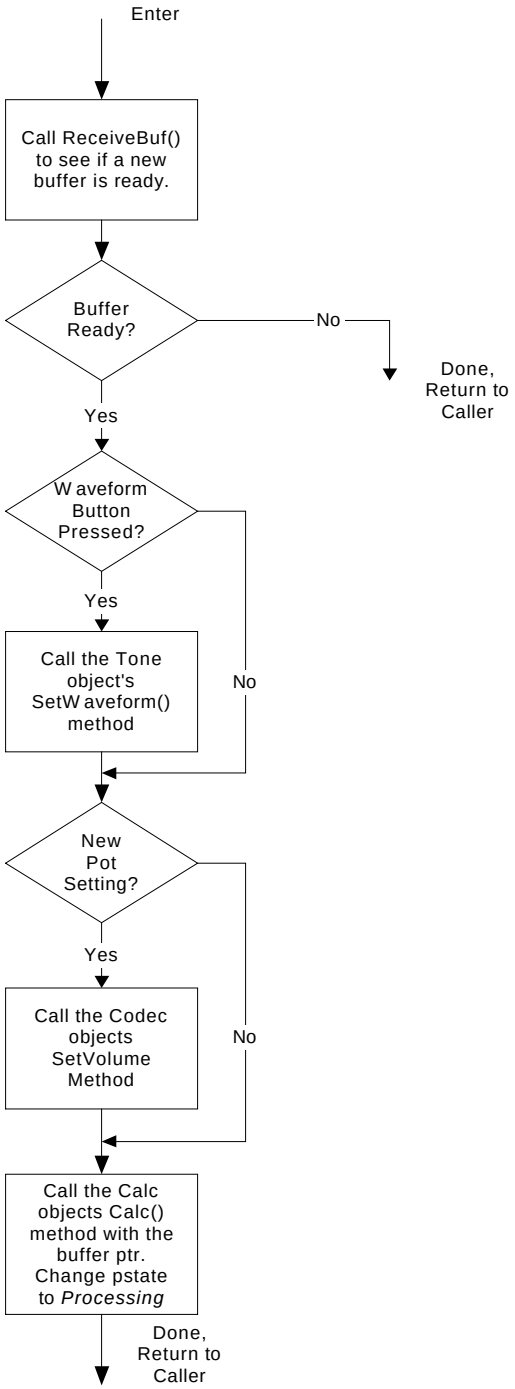
The activities in the individual states are:

START -- This is the state to which the Process() state machine is statically initialized. It does nothing and transitions to no other state. When InitProcess() is called, it changes pstate to *Idle*.

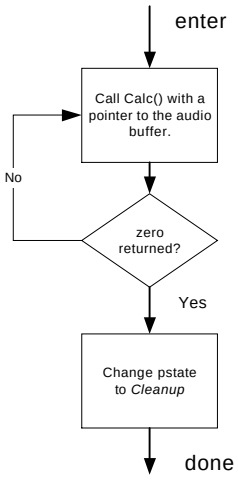
IDLE -- Each time it is called In the *Idle* state, Process() executes the flowchart shown below:

Logic of the Process() Idle State:

Each time through the process Idle state, the flowchart at the right is executed.

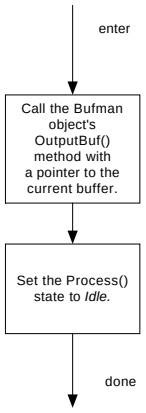


PROCESSING -- Each time it is called In the *Processing* state, Process() executes the flowchart shown below:



Process()
Processing
State

CLEANUP -- Each time it is called in the Cleanup state, Process() executes the flowchart shown below:



Process()
Cleanup
State

Verification:

1) Verify, by stepping through with a debugger, that pushing a waveform button, results in Process() calling the Tone object's SetWaveform() method before calling Calc() for the next buffer.

2) Verify by stepping through with a debugger, that setting the volume potentiometer results in Process() calling the Codec object's SetVolume() method before calling Calc() for the next buffer.

Class Name: tone

The object of the tone class inserts into the output buffer indicated, a tone that is equal in magnitude to the volume supplied, has frequency equal to the frequency supplied, and has the current output waveform.

Cardinality: 1

Source Modules:
tone.h, tone.c

Data Elements:

Name	*Type*	*Purpose*
sineTable	s16 [ ]	Waveform table for sine waveform
triangleTable	s16[ ]	Waveform table for triangle waveform
sawTable	s16[ ]	Waveform table for saw waveform
squareTable	s16[ ]	Waveform table for square waveform
w	SINE, TRIANGLE, SAW, or SQUARE	Waveform to use

Public Methods:

Name	*Return Type*	*Arguments*	*Purpose*
InitTone()	none	none	Initialize the waveform tables with computed values.
ToneInject()	none	BUF *ptr, s16 vol, s16 frequency	Injects into the buffer supplied, a tone with the volume supplied, having the frequency supplied.
SetWaveform	none	SINE, TRIANGLE, SAW, or SQUARE	Set the waveform to use to the value supplied.

Verification:
1. Verify with a scope that the output waveforms have the frequency and shape specified in the arguments to the ToneInject() function. Verify with a scope that the amplitude of the output varies as expected with the volume supplied.

Class Name: Voice

This object makes a local copy of the input buffer, low pass filters it, and then passes its samples one by one to the pitch period estimator (pestr). Then it decides whether the buffer contains a voiced or unvoiced signal, and if voiced, what the pitch period is. This object is primariliy just a shell through which calls may be made to the pitch estimation object pestr. This could change if it becomes necessary to incorporate frequency domain pitch measurement techniques to overcome limitations of the time-based approach.

Cardinality: 1

Source Modules:

voice.c voice.h

Data Elements:

Name	*Type*	*Purpose*
avgAbsMag	int	average absolute magnitude of the signal in the input buffer.
leftChan	int[ ]	array containing all of the samples from the input buffer, in which the low pass filtering is performed.

Public Methods:

Name	*Return Type*	*Arguments*	*Purpose*
InitVoice()	none	none	Currently does nothing, but was included in case pitch period estimation or the voiced/unvoiced decision algorithm is enhanced in a way that requires initialization.
CopyBuffer()	int	BUF *ptr	Copies the input buffer to the leftChan buffer, computing as it does so the average absolute signal magnitude.
FilterBuffer()	void	void	Passes the leftChan buffer to the low pass filter routine in the filter object.
EstimatePitch()	int	void	Passes each sample in leftChan buffer to the pitch estimator routine PestrNextSample(). Then it calls the pitch estimator routine which returns a pitch estimate, GetPestr().

Verification:

1. Compate the avgAbsMag values against the input buffers to make sure they are plausible.

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 be allowed to attend. All others, especially management and financial stakeholders, should be barred from attending, if that is possible.

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 almost a trivial activity. Nevertheless, preparation for coding sometimes plunges the development team into devisive conflicts. No two software engineers write code in exactly the same way. Each engineer has his own preferences and cognitive style. The two most likely conflicts are: Choice of Language and Coding Standards.

Choice of Language

Most embedded systems are written in 'C', but there are attractive alternatives. C++ is thought by many to provide better tools for translating object-oriented design concepts into source modules. Java is also a desirable language, primarily because it is a 3rd attempt at an object-oriented, C-like language.

The author's view is that C++ is better suited to large systems with many objects of similar classes, such as windowing systems, than it is to embedded systems. Java still has issues with compilation and with library size that might cause performance or code size hits in small systems.

Contrary to widely held belief, object-oriented designs need not be implemented in object-oriented languages. Object-orientation is far more important in the design phase than it is 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 necessity for writing strictly typesafe code in an object-oriented language often complicates the implementation of designs involving function tables and registration-callback patterns.

If you are able to use 'C' in your embedded project, I would recommend doing so. If there is significant pressure to use another language, such as C++ or Java, 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 C++. It's good practice for the day when you are called upon to implement mainframe or PC code.

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 appearance, and those who prefer broad lattitude for personal expression.

The sticklers want the appearance of source code to be tightly controlled. The manner of nesting braces must be rigidly controlled to a preferred scheme. Capitalization or non-capitalization of variable and function names must be

subject 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 baccept coding standards that affect the object code, but purely stylistic issues are seen as invasion of their personal perogative.

This conflict is, I think, mostly 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 getting involved in divisive conflict. If you are a member of a team that has chosen a coding standard that doesn't fit your *weltanshaung,* 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 voice substitution code, two languages are used. Basic is the language used in the Basic Stamp2 processor that implements the AuxInput object. On the DSP board, C is used.

Source Code for Voice Substitution Device

Appendices A and B contain the voice stress detector source code for the Basic Stamp2 and DSP processors.

DEBUG

Debugging presents a continuing sequence 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.

Catagories of Bugs

Most bugs fit into one or more catagories. It helps to bear in mind these catagories 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, many of which were represented in the voice substitution device 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. For instance, in an early prototype of the voice substitution device, a wicked memory corruption bug arose from the sequence:

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

The problem here was that the post incrementation took place after the limit check on k, but before the assignment. The fact that the post incrementation took place on the right hand side of the statement lulled my foollish brain into ignoring this fact. The result was a variable one word past the end of x[ ] was modified by the assignment. That word happened to be a pointer, resulting in trashing vast quantities of memory.

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 member 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. I used to assume that just meant bullshit section. After encountering bugs in the BSS, I learned to treat it with more respect. Now it is the Binary Storage Section. It contains the locations for all of those variables which are not consts 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 her trace the circuit malfunction.

Programmed 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 twenty 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.

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 overwhelming probability is that the software is at fault.

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

When you have looked for the same bug for a long time, your mind is 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 catagories above for applicability. 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.

Cheer up. You could be having to work for a living.

INTEGRATION

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 is especially true of resource providers, who by this time, have stuck their neck out a mile and a half supplying the money to get this far.

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 at least once a week. That way the most you can lose is a week'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.

VERIFICATION

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 and 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 voice substitution device project, the suggestions for verification were included in the Class 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 the documentation of each verification test performed for the voice substitution device example.

Object: Cinit
Test Number: 1
Test Description:
Make sure the stack location is what you expect it to be, and that its size is what you need.
Test Outcome:
Looking at the link map reveals that the stack is located at 0xba, and its size is 0x800. The location is OK, and the size is way more than it needs to be. Since no other objects need the memory, let it be.

Object: Cinit
Test Number: 2
Test Description:
Use the emulator to check initialized values of variables associated with each of the objects. Also check .const values for each object.

Test Outcome:
The following objects contain variables with initializers: auxport, bufman, calc, cpu, error, filter, pestr, and procman. Some library routines do too, but they will not be checked. All initialized variables were checked at the entry to main() and found to be OK.

The following objects contain .const entries: filter, pestr, and tone. Some library routines do too, but they will not be checked. filter had 2 floating point literals which were found to have been loaded correctly. tone had 3 floating point literals which were found to have been loaded correctly. pestr had 2 floating point literals that were found to have been loaded correctly.

Object: Cinit
Test Number: 3
Test Description:
Verify that any code which is supposed to be in RAM is actually in RAM.
Test Outcome:
Code resides in the .text section, which according to the link map is located from 000054b5 and run for a length of 00001311. I ran to main, and then checked that area with a disassembler. Yes the code was where it was supposed to be.

Object: Cinit
Test Number: 4
Test Description:
if exit() is to be used by the system, make sure that the exit code is recorded whenever someone calls exit.
Test Outcome:
exit() is never called in this system. The error handler just enters a loop. By halting the debugger, the user can find out what the error code was. This is good enough for a system that is not going into production.

Object: Cinit
Test Number: 5
Test Description:
Test all the interrupt vectors to make sure they actually transfer control to where they are supposed to.
Test Outcome:
Yes, well I did run the system quite a bit during debugging, and it certainly would not have worked if the interrupt vectors weren't vectoring to the right place.

Object: AuxIn
Test Number: 1
Test Description:
Look at the serial output of the BasicStamp2 processor on the oscilloscope.

Make sure that the picture is as shown in the timing diagram. Verify that pushing the buttons makes the appropriate change in the first four bits, and that setting the potentiometer appropriately changes the next twelve bits.
Test Outcome:
This test was performed exactly as prescribed, and everything worked the way it was supposed to.

Object: AuxIn
Test Number: 2
Test Description:
Verify that the Aux Serial port on the DSP board can receive the serial message put out by the AuxIn object.
Test Outcome:
Well, we wouldn't have got very far in debugging if it couldn't.

Object: Auxport
Test Number: 1
Test Description:
Use the emulator to make sure that when a SINE, TRIANGLE, SAW, or SQUARE button is pressed, the AuxportRxInterrupt() routine is called, and it sets sinePressed, trianglePressed, sawPressed, or squarePressed to TRUE.
Test Outcome:
Checked this with breakpoints in handleInputs() private method of procman object. Worked as expected.

Object: Auxport
Test Number: 2
Test Description:
Use the emulator to make sure that when the potentiometer wheel is moved, the latestPot value is changed.
Test Outcome:
Checked with breakpoint in handleInputs() method of procman object. Turns out, when looked at on the scope, the lower four bits of the pot reading were changing quite frequently. Modified the potChanged() routine to only return TRUE bit(s) above the lower four changed. When that is done, the latestPot value is changed only when the potentiometer wheel is moved.
.
Object: Auxport
Test Number: 3
Test Description:
Step through the SinePressed(), TrianglePressed(), SawPressed(), SquarePressed(), and PotChanged() routines to verify that they set their associated flags to FALSE.
Test Outcome:
This was done in Auxport test number 1.

Object: Bufman
Test Number: 1
Test Description:
Make sure that buffers added to the process list with PutBuf() are always received by ReceiveBuf() in the same order that they were put. Make sure that buffers added to the output list by OutputBuf() are always received by the caller of GetBuf() in the same order in which they were added.
Test Outcome:
This will be proved if the buffers put by PutBuf() aways have sequency numbers one greater than the last buffer put by PutBuf(), and if the buffers received by ReceiveBuf() always have sequency numbers one greater than the previously received buffer; and if the buffers sent with OutputBuf() always have sequency numbers one greater than the previous buffer sent with OutputBuf(), and if the buffers received by GetBuf() always have sequency numbers one greater than the buffer previously received by GetBuf(); and if the first buffer processed by each station is 0. Checks were placed in the PutBuf(), ReceiveBuf(),OutputBuf(), and GetBuf() routines to verify that these conditions are met. System was run for an hour and there was no instance of buffer counts out of sequence.

Object: Bufman
Test Number: 2
Test Description:
Verify that during normal operation, the totalcount read from each buffer increases monotonical.
Test Outcome:
The totcount field of each buffer was examined in four places as it traveled through the system. At no time did two successive buffers have totcount fields differing by anything but +1. That is a monotonicaly increasing count.

Object: Bufman
Test Number: 3
Test Description:
Verify by adding a temporary delay to the buffer processing, that if more than two buffers are used, that they continue to arrive at the output in the correct order, as shown by the totalcount field of the buffers
Test Outcome:
Extra buffers were added to the pool, and the code was changed to make sure that all of the extra buffers were in either processing list or the output list. In spite of the fact that these lists now contained more than one buffer, the buffers continued to arrive at the output in the correct order. .

Object: Bufman
Test Number: 4
Test Description:
Stop at a breakpoint after the system has been running for a while, and verify that you can account for the whereabouts of all your buffers. Do they all appear in either the free list or one of the processing lists, including inBuf, firstProcess, calcBuf, firstOut and outBuf.
Test Outcome:
This was done a bunch of time, and all buffers were accounted for.

Object: Calc
Test Number: 1
Test Description:
Verify that the potentiometer setting varies the volume from silence to a usable volume in the earphones, or a speaker if one is attached.
Test Outcome:
Hooked up earphones and then a speaker. Potentiometer setting varied both from silence to a listenable volume.

Object: Calc
Test Number: 2
Test Description:
Verify that the input of a voiced audio signal results in the output of a voiced audio signal with the waveform selected by one of the waveform buttons.
Test Outcome:
Yes, it happens that way every time.

Object: Calc
Test Number: 3
Test Description:
Verify that sweeping the input voice frequency smoothly from 50Hz to 880Hz sweeps the output frequency smoothly over the same range.

Test Outcome:
Well the frequency seemed to match quiet well during the entire sweep, but between 74 and 84 Hz there was a glitch in the phase of the sine wave. When I set the output waveform to a square wave, it appeared as a sine wave below 74Hz and a square wave above 84 Hz. The saw and triangle waves switched from sine wave below 74 Hz to the correct waveform above. Hmmmmm....

The switch in waveform was due to a casting error in the routine which matches period estimates, a private routine of the pestr object called countMatches. Once that was fixed, there remained a glitch in the phase of the output sine

wave that happened at most frequencies below 90 Hz. At 48000Hz, there are 533 samples in one cycle, which is more than four times the length of the waveform tables. Something is screwing up the phasing of samples copied out of the waveform tables.

Rewrote the tone injection routine to use a scaled integer phase that varies between 0 and 16284. That fixed the phase glitching described above.

Next it became apparent that the pitch estimating routine generates pitch estimates half what they should be at 100 and 116 Hz with a sine wave input.

Tightening up the ratio check in the period matching routine in pestr fixed the problem at 116 Hz, but the problem remained at 100 Hz.

That problem turned out to reside in the peak-to-previous-peak period estimator, whenever a pure sine-wave was input. The period estimate never got updated because the difference value seldom exceeded the threshold. That period estimate had a tendency to be twice the actual period on occasion, and since it was rarely updated, it contributed to a condition where twice the period was chosen most often.

Fixed the problem by assigning a random period below the minimum acceptable period whenever the assigned value of the estimator was zero.

Object: Calc
Test Number: 4
Test Description:
Verify with speech input that the voiced portions of the speech are reproduced with the waveform changed, but that the unvoiced portions of speech are merely copied to the output.
Test Outcome:
I spoke into the system and collected digital scope traces of both the input to the system and the output from it. The voiced portions of speech had the same envelope for both input and output, but the waveform of the output was the one selected by the waveform switch. For fricative sounds, like Sh and Th, the output was delayed slightly from the input, but its waveform was not changed.

Object: Calc
Test Number: 5
Test Description:
Verify that there is at most a single buffer period (1/200th second) delay between the input and output signals.
Test Outcome:
Looked at a bunch of scope traces showing both the input and output signals. The definitely showed that there is typically a 10msec delay (1/100th second) delay between the input and the output signals. This is possibly due to the fact

that there is a simple IIR in the period estimation algorithm. It's not a problem, so don't try to change it.

.

Object: Codec

Test Number: 1

Test Description:

Use a function generator to generate square waves, ramps, and sine waves, and verify that on input, the data in the input buffer has the same form as that generated.

Test Outcome:

The data in the buffers matches the waveforms selected on the oscillator.

Object: Codec

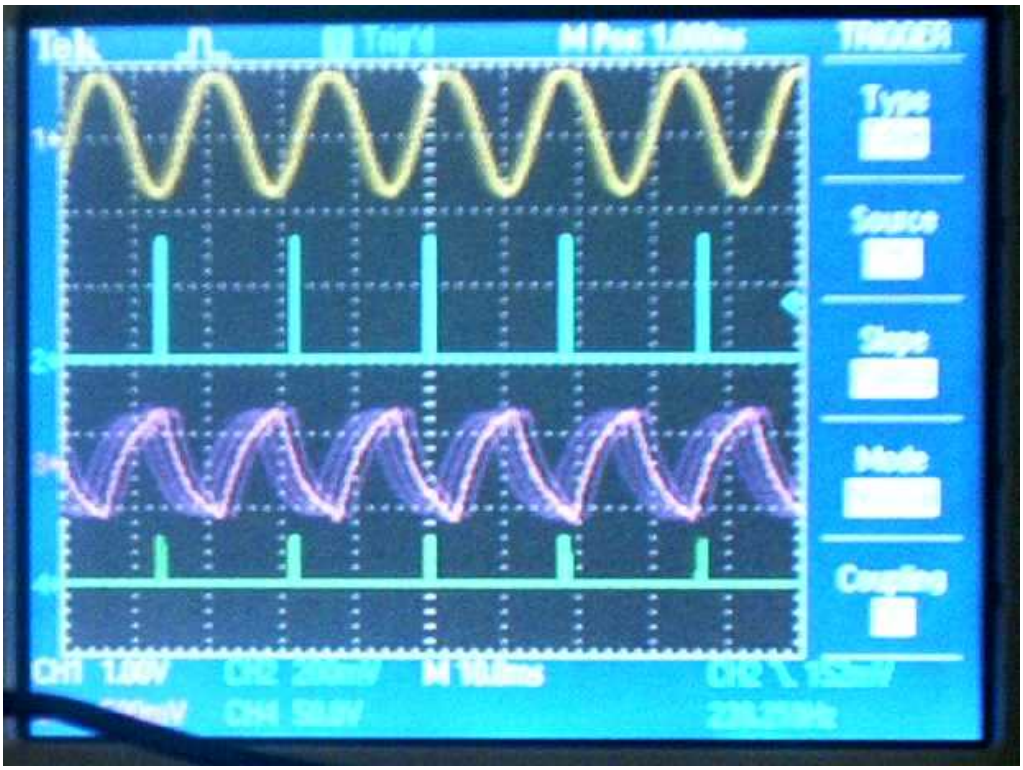
Test Number: 2

Test Description:

Use a math routine to fill up output buffers with square waves, ramps, and sine waves. Use an oscilloscope to verify that that is the wave shape that comes out of the the codec.

Test Outcome:

Sure enough, triangles, squares, and sines all came out as programmed. The picture below shows a sine wave input on top, and a triangle wave output on the bottom.



Object: Cpu
Test Number: 1
Test Description:
Run a test routine to verify that interrupts never happen while they are supposed to be disabled.
Test Outcome:
Ran a routine that disabled interrupts, set a flag, did a bunch of stupid stuff, reset the flag and enabled interrupts. The interrupt routines were modified to check to see if that flag was set. If so they would create a fatal error. Ran the program for a long time, but never saw the fatal error.

Object: Cpu
Test Number: 2
Test Description:
Look at the system clock with an oscilloscope, and verify that it is 160MHz.
Test Outcome:
Looked at the signal and it had 16 cycles every 25nsec, which corresponds to 160MHz.

Object: DMA
Test Number: 1
Test Description:
Measure the maximum interrupt latency in the system. Use the measured value to recompute the time available to each DMA interrupt. Verify that a DMA interrupt, including its calls to Bufman, always executes in less than the time available.
Test Outcome:
Rigged the code to turn an LED on whenever interrupts are disabled. Used a pulse triggered scope channel to document the longest period of interrupt disablement to be 1.27 microseconds, which is the maximum time it takes a DMA interrupt to complete. According to the notes for the Dma object, 10 microseconds is available, worst case, for each DMA interrupt to complete.

Object: DMA
Test Number: 2
Test Description:
See tests 1) and 2) for the codec object.
Test Outcome:
The fact that tests 1 and 2 of the codec object are passed shows that the DMA interrupts are happending as required. The quality of waveform indicates that no data is lost during the DMA interrupts.

Object: DMAPort
Test Number: 1
Test Description:
Hook up a scope to the Codec's left/right clock, bit clock, data in, and data out signal lines. Make sure the signals look as they should according to the datasheets, and the nature of the data being transferred.
Test Outcome:
Did that. They looked OK.

Object: Error
Test Number: 1
Test Description:
Seed the system with errors of various types, and verify that the system stops with the appropriate error code in the HandleError() routine.
Test Outcome:
Didn't have to seed the system with errors. There were plenty of errors during debugging, and the HandleError() routine worked as advertised.

Object: Filter
Test Number: 1
Test Description:
Currently the low pass filter has a cutoff of 960Hz.
Modify it to do no filtering whatsoever. What effect does this have on the operation of the device. There seem to be a couple of possibilities: either the computed voices are noiser representations of the incoming voices, or more time is consumed in computing high frequency peaks. The latter effect could cause the device to fail to meet its output deadline and crash.
Test Outcome:
The output signal was a bit noisier without the filter, but the device didn't crash.

Object: Filter
Test Number: 2
Test Description:
Modify the LowPass() routine to cut off at half of its current cutoff frequency (480Hz). What effect does that have on the operation of the device?
Test Outcome:
The pitch algorithm only returned a single pitch, regardless of the pitch of the vocal input. Subsequent tests indicated that this may have been the result of a 60Hz noise artifact that occasionally showed up.

Object: Filter
Test Number: 3
Test Description:
Modify the LowPass() routine to cut off at 720Hz.
Test Outcome:
The pitch algorithm still works but it is not as noisy as with the 960Hz cutoff. Leave the cutoff at 720Hz.

Object: Main
Test Number: 1
Test Description:
This object will be thoroughly tested when debugging the rest of the system is complete.
Test Outcome:
And so it has been.

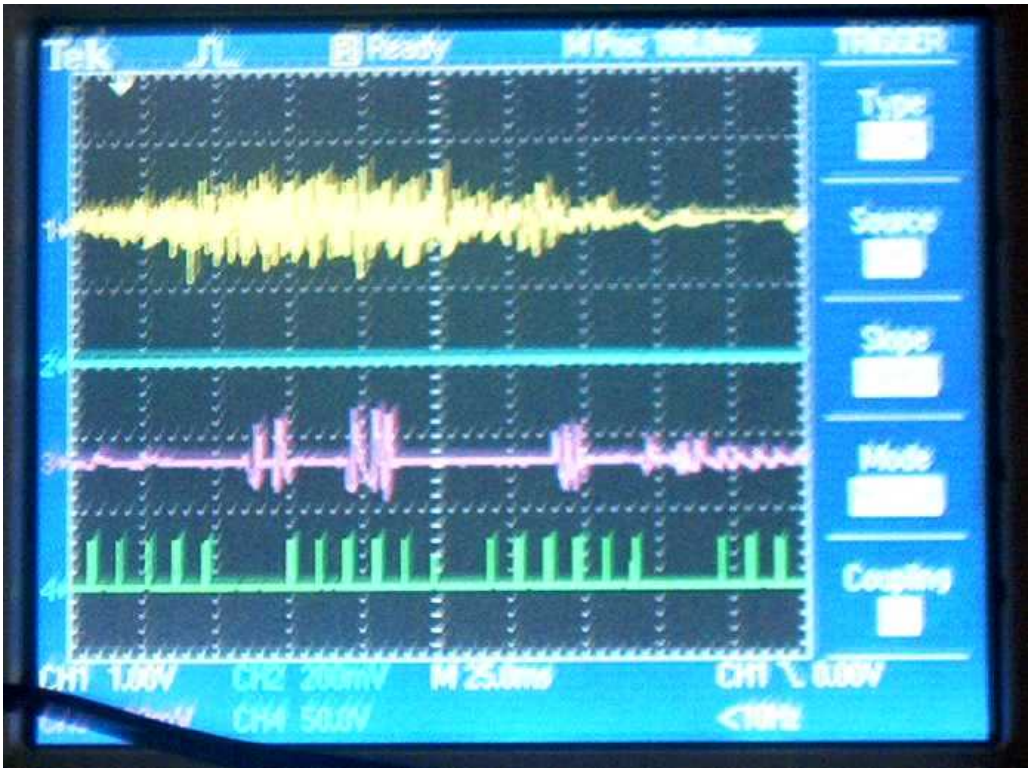
Object: Pestr
Test Number: 1
Test Description:
Run the algorithm with sine wave, square wave, saw, and triangle wave inputs in the frequency range of 50-880 Hz. Verify that it correctly identifies the pitch frequency in all cases.
Test Outcome:
Well, its always within 1% of the pitch anyway. Guess that's going to have to be close enough.

Object: Pestr
Test Number: 2
Test Description:
Run the algorithm with voice input from a microphone. Gradually sweep the voice from low range to high range and back, and verify that the algorithm tracks the pitch changes.
Test Outcome:
Yes, it tracks close enough that you can't hear the difference between the pitch of the input and the pitch of the output.
.

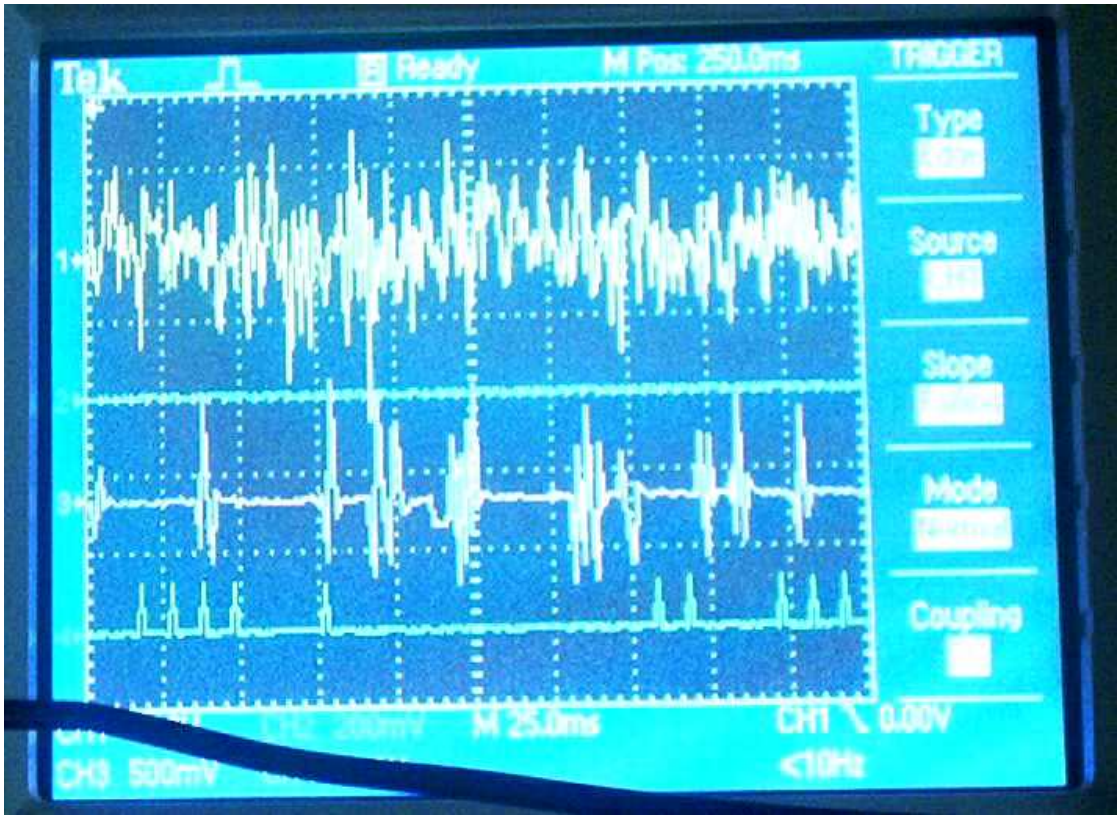
Object: Pestr
Test Number: 3
Test Description:
Using spoken input, verify that a no-pitch decision is reached for hissing. Use a white noise source to verify that a no-pitch decision is reached for white noise input.
Test Outcome:
I modified Calc() routine to inject silence when a no-pitch decision is reached.

Then made various voiced sounds, and unvoiced hissing sounds into the microphone.

Hissing into the microphone causes a large reduction in the number of voiced decisions. The waveforms produced are sparse compared to the waveforms resulting from voiced input. Upper trace input, lower trace output.



Next, I used an envelope generator to gate white noise to the input and observed the output wave from whenever the gate was operating. The output waveforms were sparse compared to the input waveforms. Upper trace input, lower trace output.



The no-pitch decisions are not continuous for hissing and white noise, but they do predominate. This is probably good enough.

Object: Procman
Test Number: 1
Test Description:
Verify by stepping through with a debugger, that pushing a waveform button results in Process() calling the Tone object's SetWaveform() method before calling Calc() for the next buffer.
Test Outcome:
Actually it is the routine handleInputs(), called from idleState(), called fromProcess() that actually calls SetWaveform().

Object: Procman
Test Number: 2
Test Description:
Verify by stepping through with a debugger, that setting the volume potentiometer results in Process() calling the Codec object's SetVolume() method before calling Calc() for the next buffer.
Test Outcome:
Actually it is the routine handleInputs(), called from idleState(), called from Process() that actually calls SetVolume().

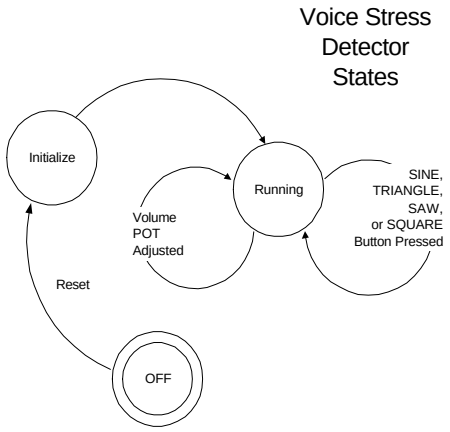
Object: Tone
Test Number: 1
Test Description:
Verify with a scope that the output waveforms have the frequency and shape specified in the arguments to the ToneInject() function. Verify with a scope that the amplitude of the output varies as expected with the volume supplied.
Test Outcome:
These facts were verified numerous times in the course of debugging.

Object: Voice
Test Number: 1
Test Description:
Compare the avgAbsMag values against the input buffers to make sure they are plausible.
Test Outcome:
Sine wave peaks were around +/- 32000 with absAvgMag values in the vicinity of 19700. The ratio is 0.616 which sounds pretty close for the average absolute value of a sine wave which would be 0.625.

VALIDATION

Verification is a double check that the system's virtual objects perform as they are supposed to. Validation is a series of checks to verify that the system as a whole meets its requirements. The validation report for the voice stress detector 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) When powered up, the voice substitution device briefly enters an initialization phase, during which it sets up the system hardware and computes waveform tables. Then it enters normal operation using full volume, and a Saw waveform as its defaults.

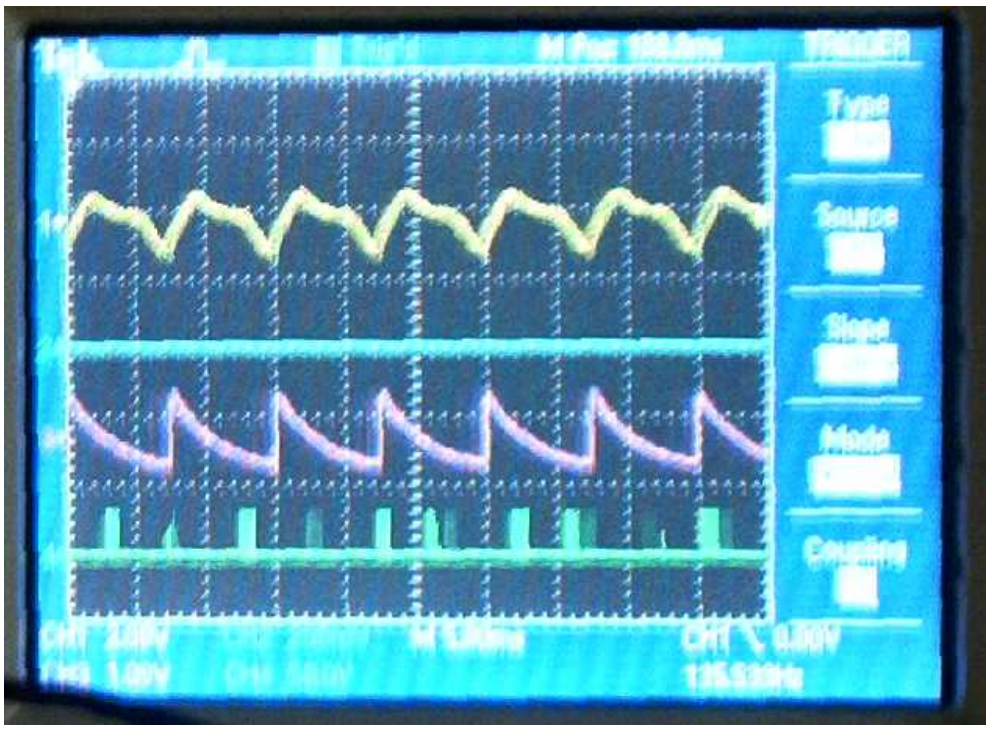


The behavior specified in this requirement has been observed with the Code Composer Studio debugger.

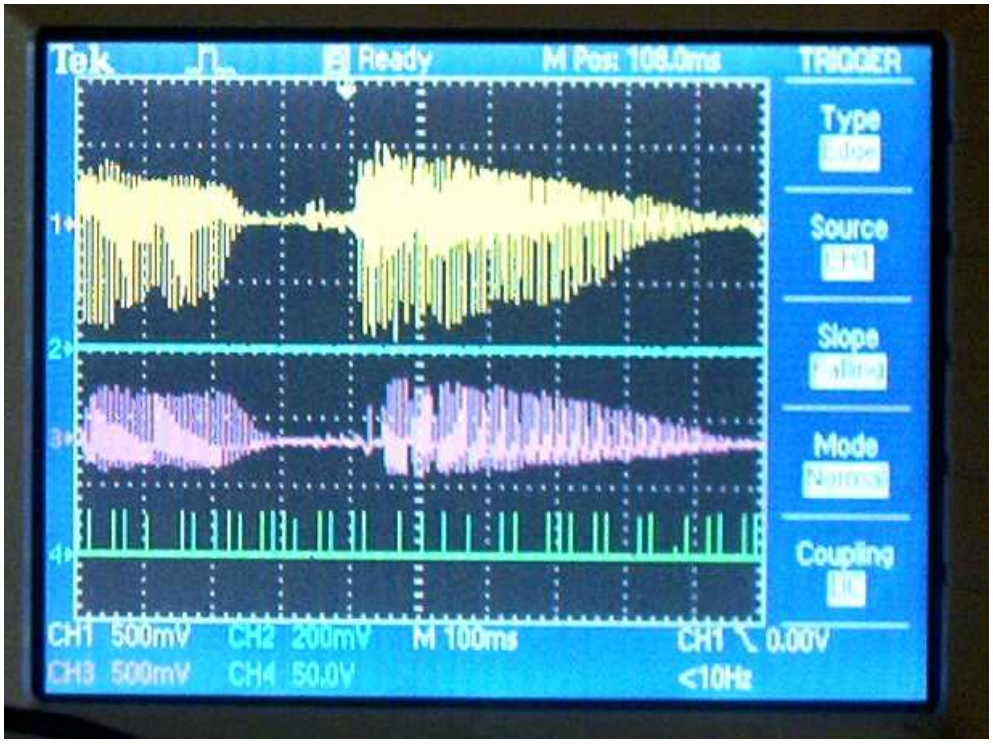
2) The voice stress detector accepts a microphone or line level input signal. It outputs an audio signal to the earphone or line level outputs, which consists of an echo of the input signal with voiced sounds replaced by a synthesized waveform.

The system behavior specified by this requirement has been repeatedly observed during testing and debugging. See the next page for examples.

This scope picture shows an example of microphone input in an upper trace, with the synthesized saw wave output in a lower trace:



The picture below uses a different timebase, showing the copying of the envelope of the input signal to the output signal.



3) When no voice is detected, the input signal is merely copied to the output without modification to the waveform, except that the volume may be reduced if that is requested by the potentiometer setting.

Copying of unvoiced input directly to output with attenuation as directed by the volume potentiometer was observed to work correctly.

4) The voice stress detector shall monitor four hardware switches called SINE, TRIANGLE, SAW, and SQUARE, as well as the setting of a potentiometer.

This behavior was repeatedly observed to be working correctly.

5) The hardware switches shall enable the user to choose between four different output voice waveforms, sine wave, triangle wave, saw wave, or square wave.

Yes, one can choose between output waveforms using the switches.

6) The potentiometer setting shall control the volume of the output, from silence to whatever is the volume of the input.

Volume control works as required.

7) Voiced portions of the signal shall have a fundamental output frequency equal to the fundamental input frequency determined by the voice detection algorithm. The frequency match shall be close enough to avoid obvious disharmony between the input and the output signal over the frequency range in which the device operates.

This was demonstrated during verification testing.

8) The voice substitution device shall operate within the frequency range of 48Hz to 880Hz.

This was demonstrated during the course of debugging and validations testing.

9) The device is expected to operate correctly with vocal input from a preamplified microphone. Operation from noisy, off-air or taped sources is not required.

Actually, the gadget works from noisy, off-air sources, but the pitch-detection algorithm tends to be a bit qwirky.

10) The device shall be capable of recomputing a voice/unvoiced decision and voice frequency estimate at two hundred times per second.

It renders one voiced/unvoiced/pitch decision for each buffer passing through

the system. The buffers pass through at the rate of two hundred times per second.

11) The device shall introduce an envelope delay of no more than 5.0 msec between the input and the output signals.

This is a more complex subject than this requirement implies. The output volume setting tracks the input volume within 5-25msec depending upon the IIR filter used to smooth the volume setting. That filter is necessary because at low frequencies, one cycle of a waveform spans several buffers. Without the filter, single buffer volume measurements cause output waveform distortion.

Another issue is the speed with which the period estimate comes up with a voice decision. Although it makes a new decision every buffer (5msec), it sometimes takes a few buffers for it to notice a voice.

So neither the volume envelope, nor the pitch estimation tracks the signal in anything like 5.0msec. If this requirement can be relaxed, all the better. If not, we will have to find faster algorithms for tracking envelope and pitch. Perhaps something based upon Hilbert transform pairs would do a better job.

Since the author is the end user of this device, he has the discretion to waive this requirement. In the interests of finishing the book, that is just what he chooses to do.

Summary of Validation Results

The performance of the pitch detector is adequate for the purpose of this project, which is to serve as an example for a book. It is also adequate for use by audio hobbyists and musicians who are looking for an interesting effect to exploit.

On the other hand, there is enough noise in the pitch detection algorithm that this particular device will never be used to create sounds for an easy-listening radio station. Addition or substitution of other, smoother methods of pitch detection is recommended for commercial audio use. Such methods might include Cepstral analysis or a short time auto-correlation approach.

Such activities are left as a spare time exercise for the author or the reader.

DEVELOPMENT ENVIRONMENT

People live and work in a more or less supportive environment. The more supportive the environment, the more attention people can devote to their work. The less supportive the environment, the more time must be spent dealing with personal needs, and conflicts arising from those needs. This chapter describes several environmental factors that affect the outcome of embedded projects. These factors include: tools, communication, management, and stress reduction behavior.

TOOLS

It is possible to blow a huge bundle of project funds on tools, and get nothing in return. This is a sad fact, but in today's world a true one. There are many tool vendors out there who make extravagant claims, and deliver only paltry assistance. The best advice one can give those responsible for tool choice is to make your own tools where possible, and buy only from long established tool vendors where necessary.

That being said, there are some tools that you cannot get along without.

You're going to need a good **word processor** and **diagramming tool** for use during the analysis and design phases. In PC land, Microsoft makes a good word

processor, Word. SmartDraw.com makes a good diagrammer. In the land of Linux, Star Office is a good word processor, and Dia is an adequate diagrammer.

I have found it advisable to avoid the use of so-called **CASE tools** for documenting architecture and design work; because they require rigorous adherence to design rules that are compatible only with their favored (frequently overblown) methodology.

When you start writing code, a good **text editor** is your best friend. Everyone has their favorite. Mine is Visual SlickEdit. It provides source code management, build support, and debugging support that can be individually configured for each project. It also has a good FTP client, and support for version control system access. It is available on Unix, Linux, Windows, and IBM mainframe operating systems.

You can't do without **debuggers**. They can be had for free, or for tens of thousands of dollars. The value of the expensive ones is not that much greater than the value of the cheap ones, unless they can double as bus state analyzers for FLASH, RAM and EPROM, which most can't. The GNU tool chain is available for many microprocessors, and usually has adequate debugging support. For expensive in-circuit emulators, with fancy features, insist on trying the unit in your own environment before you buy.

An **oscilloscope** is vital for everyday debugging, watching I/O ports and looking at two or three wire serial busses. Tektronix has some four channel, color LCD scopes, with built-in spectrum analysers, which can be had for surprisingly low prices.

A **logic analyzer** can be helpful in tracking down hidden bugs. I prefer the products of Agilent (formerly HP), but almost any moderately priced one will do. Most medium size companies will have one or more logic analyzers laying around. Spend some time and learn how to use it. Sometimes it is your best hope for finding a tricky hardware or software bug.

For the voice substitution device project, an old-fashioned **analog synthesizer** from Synthesizers.com was used for trying out algorithms and verifying the final operation of the device.

COMMUNICATION

Many people choose engineering because they think they are more interested in "things" than in people. It is easier for them to communicate with their computers than with their parents, siblings or friends. Ironically, successful engineering projects depend totally upon the interpersonal and communications skills of the engineers.

Where communication between team members is easy and pleasant, ideas flow freely, and the project makes progress. Where communication is strained due to professional conflicts, personal conflicts, or fear of the resource provider, progress is stifled.

Story #1

Once I was assigned to develop an operating system for an embedded project. The project appeared to be foundering because the person assigned to the operating system was not delivering his work on time. On my first day on the job, this man would only glower at me. He would not speak to me.

I began to bribe him to talk to me by giving him homemade chocolate chip cookies. Over a period of a week, we gradually began to speak. It turns out he had much more on his mind than the project. He soon went on to other activities. When I examined his code, it turned out to be partially usable, with a number of innovations that I have used ever since.

Story #2

On another occasion, I worked with a man who so rubbed me the wrong way, I could hardly stand to occupy the same room with him. I soon made the feeling mutual. Not only that, but we had opposite views on the best way to proceed in the project. Before the project was over, I discovered that I liked the man personally (his irreverence matched my own... he just applied it to my work), but I never agreed with his technical approach. That relationship took a big chunk of my time and energy, and negatively impacted the architecture and design processes. By the time integration came around, I actually allowed him to contribute to my part of the project. That was a great accomplishment. It was made possible by the intervention of a couple of good managers.

Whatever you do in an embedded project, you must attempt to establish actual friendly communication with every other member of the team. That way you won't be avoiding people in the halls or wasting time in needless arguments. You will be maintaining good boundaries, not suffering from abuse at the hands of sometimes cretinous co-workers.

MANAGEMENT

It is hard to predict how long an embedded software project will take, or how successful it might be. This makes project management difficult. Not only does the manager have to work with software developers who can behave more like quarreling cats than professional engineers. He also must survive at the bottom of a primate dominance hierarchy, in which people occasionally act more like baboons than professional managers.

One thing you can do for your manager is invite her to participate in your bizarre behavior (See section on Remedies to the Obstacles of burnout and stuck mind). It gives her a break from being serious for a while, and makes her feel less alone.

Often, the attitude of your manager is the best clue you have to the attitude of the ultimate resource provider for your project.

If your manager is engaged in the project, takes notes at meetings, and appears to care what is happening, things are probably going well at the higher levels.

If your manager shakes his head and rolls his eyes inward, spends a lot of time with the scheduling program, and seems a bit too desperate to have a good time at meetings, he is probably looking for another job. That is not a good sign for the project.

The more managers you have on a project, whether serially or in parallel, the less likely the project is to succeed.

All in all, it is best to take good care of your manager. Go out of your way to supply her with project estimates, and updates on your progress. Keep her informed of any technical difficulties you may be having, or any conflicts you may be suffering with other team members.

Above all be thankful for your manager. She is the only thing standing between you and that barbaric, primate dominance hierarchy.

OBSTACLES

Embedded software projects are hard enough in their technical aspect. Then most of us complicate them with petty jealousies, and dislikes of our co-workers and managers. Combine these factors with unrealistic deadlines and excessive demands on your personal time, and you have a recipe for burnout.

Burnout

Burnout is a condition that can be reached where you simply have invested too much energy in your work, and lost all sense of proportion

and balance in the rest of your life. You are physically exhausted, riddled with anxiety, and want nothing more than to spend a few years hanging out at the head waters of the Ganges, or sipping Pina Colada's on the beach of some Caribbean resort. It's best not to let things go that far; because it will take you a minimum of six months, and more likely a year to recover.

Even if an engineer doesn't burn out under stress, he can waste countless hours of project time beating his head against the brick wall of mentalis immobilus, otherwise known as Stuck Mind.

Stuck Mind

Sometimes it seems the harder we try to solve a problem, the more intractable it becomes. This is especially true when we are under stress, or experiencing fear. Problem solving requires a playful, supple mind. When negative emotion is active, it hardens the mind, and blinds it to helpful clues.

REMEDIES

Since both burnout and stuck mind result from driving the mind too hard in the presence of negative emotion, the treatment for both conditions is similar: Have fun. Below are listed some ways to puncture the stress of an embedded project, and to relax the mind long enough for it to release its death grip on the problems and miseries of work.

Athletics

In one company, the engineers and others played frisbee football at lunch. Actually, it looked more like frisbee rugby, such was their enthusiasm for the game. When they came back to work after a hard game and a hot shower, they were peaceful and focused for the rest of the afternoon. Curiously, they never seemed to get out of their chairs on those afternoons.

Massages

When your environment is mentally and emotionally stressful, attention to your body is what relaxes you. Some companies provide Massurs and Massuses to give 15 minute massages during the day.

Bizarre Activities

When your mind is stuck, you need to alter something in your life to get it to unstick. The weirder the action you can engage in, the more likely it is to unstick your mind.

One afternoon, I could not get an EPROM programmed. After five

attempts, I was still baffled by the user interface of the programmer, and unable put my code into the chip. For some reason, I chose to stand on one leg in a chair, hold up a stick of EPROMs, and sing the national anthem. After that, the EPROM burner responded instantly to my commands, and programmed the chip.

Cube Wars

This probably belongs under the heading of Bizarre Activities, but it is an important enough activity that it deserves its own heading. Rubber band fights and spitball wars are not uncommon in cube farms. More rare, but also more fun, are water gun fights, or water cannon fights. New desktop robots and/or tanks can be set against each other in tests of strength and stability. All these activities take your mind off of the insoluble problem at hand. More importantly, they lead team members to confront each other directly, instead of swapping guarded comments at meetings. A little mock combat can lighten up a project team. Too much can lead to trouble.

Wild Partying

In one project, the technical leader decided that we would have a smashing party on the day we delivered the final product. That day was so long in coming that we started taking afternoons off for so-called practice parties. These usually involved controlled substances, music, and general loose behavior. They were intended as warm-ups for the big party that kept receding into the future. As a result, all of us on the team became very close friends, and were finally able to deliver the product. We never did hold the official delivery party, for which we had been practicing.

Vegging Out

The secret to vegging out at work is to find a quiet place that no one else knows about. If you are lucky, it may be in a nearby park. It could be on the roof of your building. If you have an office, you can close the door and lie down under the desk. Find a place, and go there when you feel the need to get away from work or your co-workers. Just sitting quietly for a few minutes with eyes closed can be a big help on a stressful day.

Meditating

Meditating is similar to vegging out, except that a portion of the mind is detached from the rest, and simply observes what the rest of the mind is doing. It takes a bit of practice to master the technique. When not taken too seriously, it can be an enlightening experience. It will definitely help release the built up pressure of a bad day.

APPENDIX A -- Basic Stamp2 Code

The Auxin object is implemented on the Basic Stamp2 activity board. The class catalog entry of the Auxin object gives a relatively detailed description of how the code must be written accomplish its mission. Fortunately, a code library is supplied with the Basic Stamp2 development system. This library made it unnecessary to do the detailed coding envisioned at design time.

The code shown below, which makes use of that code library, was all that was necessary to produce the serial bus output shown under Diagrams, Algorithms, etc. in the Auxin section of the class catalog.

```
' Auxin.bs2
' -----------------------------------------------------
' This program reads four buttons and the potentiometer
' On the Basic Stamp Activity board, and composes and
' Sends an eight bit synchronous serial message roughly
' 50 times per second. The upper four bits of the message
' are the MINUS and PLUS buttons.  The lower four bits
' of the message is the potentiometer setting (Should
' be 0 to 64, but its 0 to 58, which is close enough).
' The button bits are set whenever the state of the
' button changes from not-pressed to pressed.  It will
' be up to the message receiver to latch the fact of
' the button press, so it is not lost when the next
' bit of input is received.
' -----------------------------------------------------
'{$STAMP  BS2}
SineWrk VAR BYTE
TriangleWrk VAR BYTE
SawWrk VAR BYTE
SquareWrk VAR BYTE
Buttons VAR   WORD
Pot       VAR     WORD
Result    VAR       WORD      'Word variable to hold result
Msg     VAR     WORD    ' Word variable to hold 10 bit message
Again:
  Result = 0
  Buttons = 0
  BUTTON 8,0,255,250,SineWrk,0,NoSine   ' Go to NoSine unless P8 = 0.
  Buttons = Buttons + 1
NoSine:
  BUTTON 9,0,255,250,TriangleWrk,0,NoTriangle     ' Go to NoTriangle unless P9 = 0.
  Buttons = Buttons + 2
NoTriangle:
  BUTTON 10,0,255,250,SawWrk,0,NoSaw    ' Go to NoSaw unless P10 = 0.
  Buttons = Buttons + 4
NoSaw:
  BUTTON 11,0,255,250,SquareWrk,0,NoSquare  ' Go to NoSquare unless P11 = 0
  Buttons = Buttons + 8
NoSquare:
'  DEBUG HEX Buttons
  HIGH 7                          'Discharge the cap
  PAUSE 1                         'for 1 ms.
  RCTIME 7, 1, Result                     'Measure RC charge time.
  HIGH 3
  PAUSE 1
  LOW 3
  Pot = Result/2                 ' Convert to interval 0-192 or so
  Msg = Buttons*4096+Pot              ' Compose message
  SHIFTOUT 6,5,MSBFIRST,[Msg\16]     ' Shift 16 bits of Msg value out
 ' PAUSE (64-Pot)/5                  ' Keep reporting time relatively constant
GOTO Again
```

APPENDIX B -- DSP Code

The rest of the classes in the class catalog are implemented in 'C' or 54x assembler using TI's Code Composer Studio on the Spectrum Digital TMS320VC5416 DSK board. All of the resulting source code, excluding TI-supplied library routines, is shown in the pages of this appendix which follow.

```
/*
 * types.h
 *
 *     This is a collection of standard data types
 *  used throughout the project.
 *
 */
#ifndef TYPES_H
#define TYPES_H
typedef char  s8;
typedef unsigned char u8;
typedef short s16;
typedef short DATA;
typedef double FLOATING;
typedef unsigned short u16;
typedef long s32;
typedef unsigned long u32;
typedef char bool;
/*
 * some useful definitions
 */
#define NIL ((s16)-1)
#define NULL ((void *)0)
#define TRUE (1==1)
#define FALSE (1==0)
#define ON (1 == 1)
#define OFF (1==0)
#define PI  ((FLOATING)(3.141592654))
#endif
```

```
*****************************************************************************
*  CINIT                                                                    *
*                                                                           *
*  Adapted from boot.asm   v3.70                                            *               *
*  Copyright (c) 1993-2001 Texas Instruments Incorporated                   *
*****************************************************************************
          .file "cinit.asm"
          .c_mode
          .mmregs
CONST_COPY          .set 0
*****************************************************************************
*                                                                           *
*   This module contains the following definitions :                        **
*
*         __stack    - Stack memory area                                    *
*         _start     - Boot function                                        *
*         _var_init  - Function which processes initialization tables       *
*         _exit      - End of processing
*                                                                           *
*****************************************************************************
    .ref   cinit, pinit
          .global  _start
          .global  _main, _exit, __STACK_SIZE

*****************************************************************************
* Declare the stack.  Size is determined by the linker option -stack.  The *
* default value is 1K words.                                                *
*****************************************************************************
__stack:  .usect    ".stack",0

*****************************************************************************
* FUNCTION DEF : _c_int00                                                    *
*                                                                           *
*   1) Set up stack                                                         *
*   2) Set up proper status                                                 *
*   3) If "cinit" is not -1, init global variables                          *
*   4) call users' program                                                  *
*                                                                           *
*****************************************************************************
          .text
_c_int00:
*****************************************************************************
*  INIT STACK POINTER.  REMEMBER STACK GROWS FROM HIGH TO LOW ADDRESSES.    *
*****************************************************************************
          STM       #__stack,SP                 ; set to beginning of stack memory
          ADDM      #(__STACK_SIZE-1),*(SP) ; add size to get to top
          ANDM      #0fffeh,*(SP)               ; make sure it is an even address

          SSBX      SXM                         ; turn on SXM for LD #cinit,A

*****************************************************************************
* SET UP REQUIRED VALUES IN STATUS REGISTER                                 *
*****************************************************************************
          SSBX      CPL                         ; turn on compiler mode bit
```

```
          RSBX      OVM                           ; clear overflow mode bit

*****************************************************************************
* SETTING THESE STATUS BITS TO RESET VALUES.  IF YOU RUN _c_int00 FROM     *
* RESET, YOU CAN REMOVE THIS CODE                                           *
*****************************************************************************
          LD        #0,ARP
          RSBX      C16
          RSBX      CMPT
          RSBX      FRCT

*****************************************************************************
*  IF cinit IS NOT -1, PROCESS INITIALIZATION TABLES                        *
*  TABLES ARE IN PROGRAM MEMORY IN THE FOLLOWING FORMAT:                    *
*                                                                           *
*       .word  <length of init data in words>                               *
*       .word  <address of variable to initialize>                          *
*       .word  <init data>                                                  *
*       .word  ...                                                          *
*                                                                           *
*  The cinit table is terminated with a zero length                         *
*                                                                           *
*****************************************************************************
          .if       __far_mode
          LDX       #cinit,16,A
          OR        #cinit,A,A
          .else
          LD        #cinit,A                ; Get pointer to init tables
          .endif

          ADD       #1,A,B
          BC        DONE_CINIT,BEQ                ; if (cinit == -1) no init tables

*****************************************************************************
*  PROCESS INITIALIZATION TABLES.  TABLES ARE IN PROGRAM MEMORY IN THE      *
*  FOLLOWING FORMAT:                                                        *
*                                                                           *
*       .word  <length of init data in words>                               *
*       .word  <address of variable to initialize>                          *
*       .word  <init data>                                                  *
*       .word  ...                                                          *
*                                                                           *
*  The init table is terminated with a zero length                          *
*                                                                           *
*****************************************************************************
          RSBX      SXM                           ; do address arithmetic unsignedly
        .if         __far_mode
          .else
          NOP
          LD        #cinit,A            ; don't want this sign extended anymore1
          .endif
          B         START_CINIT                   ; start processing

LOOP_CINIT:
          READA     *(AR2)                        ; AR2 = address
          ADD       #1,A                          ; A += 1

          RPT       *(AR1)                        ; repeat length+1 times
          READA     *AR2+                         ; copy from table to memory

          ADD       *(AR1),A            ; A += length (READA doesn't change A)
          ADD       #1,A                          ; A += 1

START_CINIT:
          READA     *(AR1)                        ; AR1 = length
          ADD       #1,A                          ; A += 1
          BANZ      LOOP_CINIT,*AR1-    ; if (length-- != 0) continue
DONE_CINIT:

*****************************************************************************
*  IF pinit IS NOT -1, PROCESS INITIALIZATION TABLES                        *
*  TABLES ARE IN PROGRAM MEMORY IN THE FOLLOWING FORMAT:                    *
*                                                                           *
*       .word  <address of initialization routine to call>                  *
*       .word  ...                                                          *
*                                                                           *
*  The pinit table is terminated with a NULL pointer                        *
*                                                                           *
*****************************************************************************
          SSBX      SXM
          FRAME     -4
;         nop

          .if       __far_mode
          LDX       #pinit,16,A
          OR        #pinit,A,A
          .else
        LD      #pinit,A                ; A = &pinit table
          .endif

        ADD     #1,A,B                  ; B = A + 1
        BC      DONE_PINIT,BEQ          ; if (pinit == -1) no pinit tables
```

```
       .if     __far_mode
       .else
       RSBX    SXM                     ; do address arithmetic unsignedly
       NOP
       LD      #pinit,A                ; don't want this sign extended anymore
       .endif

        BD      START_PINIT
          DST       A, @2
          nop

LOOP_PINIT:
          .if       __far_mode
          FCALA     B                           ; call function
          .else
          CALA      B                           ; call function
          .endif

        DLD     @2, A                   ; put PINIT pointer in A

START_PINIT:
        READA   @0                      ; "push" address of function

          .if       __far_mode
          ADD       #1,A
          READA     @1
          .endif

        .if         __far_mode
          ADD       #1, A
          DST       A, @2
        DLD     @0, B
        BC          LOOP_PINIT,BNEQ

        .else
        LD      @0, B                   ; "pop" address of function
        BCD     LOOP_PINIT,BNEQ         ; if not NULL, loop.
        ADDM    #1,@3                   ; move PINIT pointer (in stack)
        .endif


DONE_PINIT:
          RSBX      SXM
          FRAME     4

*****************************************************************************
*  CALL USER'S PROGRAM                                                      *
*****************************************************************************
          .if CONST_COPY
          .if __far_mode                ; Use far calls for C548 in far mode
          FCALL     _const_init           ; move .const section to DATA mem
        .else
        CALL        _const_init
          .endif
          .endif

          .if __far_mode                ; Use far calls for C548 in far mode
          FCALL    _main
          FCALL    _exit                        ; call exit instead of abort so that
          .else
          CALL    _main
          CALL    _exit                        ; call exit instead of abort so that
          .endif

          .if CONST_COPY

*****************************************************************************
* FUNCTION DEF : __const_init                                               *
*                                                                           *
*  COPY .CONST SECTION FROM PROGRAM TO DATA MEMORY                          *
*                                                                           *
*   The function depends on the following variables                         *
*   defined in the linker command file                                      *
*                                                                           *
*   __c_load         ; global var containing start                          *
*                      of .const in program memory                          *
*   __const_run      ; global var containing run                            *
*                      address in data memory                               *
*   __const_length   ; global var length of .const                         *
*                      section                                              *
*                                                                           *
*****************************************************************************
        .global __const_length,__c_load
        .global __const_run
_const_init:

        .sect ".c_mark"              ; establish LOAD adress of
        .label __c_load             ; .const section

        .text
*****************************************************
* C54x VERSION                                      *
*****************************************************
```

```
        LD      #__const_length, A
        BC      __end_const,AEQ
        STM     #__const_run,AR2 ; Load RUN address of .const

        RPT     #__const_length-1
        MVPD    #__c_load,*AR2+  ; Copy .const from program to data

*******************************************************
*  AT END OF .CONST SECTION RETURN TO CALLER          *
*******************************************************
__end_const:
          .if    __far_mode

          .if __no_fret
          FB        _freti549
          .else
          FRET
          .endif

          .else
        RET

          .endif
        .endif

*******************************************************
*  exit code just sits here and spins                 *
*******************************************************
_exit:
    FB _exit

*******************************************************
*  interrupt vectors for processor reset              *
*  auxport,dmain, dmaout, spurious interrupt, and     *
*  spurious trap                                      *
*******************************************************
        .sect ".vectors"

        .ref _c_int00
        .ref _SpuriousInterrupt
        .ref _SpuriousTrap
        .ref _AuxportRxInterrupt
        .ref _DMAInInterrupt
        .ref _DMAOutInterrupt

        .align  0x80            ; must be aligned on page boundary

RESET:                          ; reset vector
        FBD _c_int00            ; branch to C entry point
        STM #512,SP             ; stack size of 200
nmi:                            ; non maskable interrupt
          BD _SpuriousInterrupt
        PSHM XPC
        NOP

; software interrupts
sint17                ; since it does a ret instead of a rete
          RET               ; this interrupt leaves interrupts
          NOP               ; disabled.  This is done on purpose to
          NOP               ; provide a reliable method of disabling
          NOP               ; interrupts (thank you Jim)

sint18  BD _SpuriousTrap
        PSHM XPC
        NOP

sint19  BD _SpuriousTrap
        PSHM XPC
        NOP

sint20  BD _SpuriousTrap
        PSHM XPC
        NOP

sint21  BD _SpuriousTrap
        PSHM XPC
        NOP

sint22  BD _SpuriousTrap
        PSHM XPC
        NOP

sint23  BD _SpuriousTrap
        PSHM XPC
        NOP

sint24  BD _SpuriousTrap
        PSHM XPC
        NOP

sint25  BD _SpuriousTrap
        PSHM XPC
```

```
        NOP

sint26  BD _SpuriousTrap
        PSHM XPC
        NOP

sint27  BD _SpuriousTrap
        PSHM XPC
        NOP

sint28  BD _SpuriousTrap
        PSHM XPC
        NOP

sint29  BD _SpuriousTrap
        PSHM XPC
        NOP

sint30  BD _SpuriousTrap
        PSHM XPC
        NOP

; hardware interrupts
int0:   BD _SpuriousInterrupt
        PSHM XPC
        NOP

int1:   BD _SpuriousInterrupt
        PSHM XPC
        NOP

int2:   BD _SpuriousInterrupt
        PSHM XPC
        NOP

tint:
        BD    _SpuriousInterrupt
        PSHM  XPC
        NOP

rint0:
              BD _AuxportRxInterrupt
        PSHM XPC
        NOP

xint0:
              BD _SpuriousInterrupt
        PSHM XPC
        NOP

rint2:
        BD _SpuriousInterrupt
        PSHM XPC
        NOP

xint2:
        BD  _SpuriousInterrupt
        PSHM  XPC
        NOP

int3:   BD _SpuriousInterrupt
        PSHM XPC
        NOP

hint:
          BD _SpuriousInterrupt
        PSHM XPC
        NOP

rint1:
          BD _SpuriousInterrupt
        PSHM XPC
        NOP

xint1:
          BD _SpuriousInterrupt
        PSHM XPC
        NOP

dmac4:
          BD _DMAOutInterrupt
        PSHM XPC
        NOP

dmac5:
          BD _DMAInInterrupt
        PSHM XPC
        NOP


          .end

/*
```

```
 * main.c
 *
 * A Voice Substitution Device module
 * Copyright 2003, David Clifton
 * All Rights Reserved
 *
 * This object contains the main function for
 * the voice substitution device.
 */
#include "types.h"
#include "error.h"
#include "cpu.h"
#include "auxport.h"
#include "bufman.h"

/*
 * main
 *    Initialize the CPU, DMA, Serial Ports, Codec,
 * Buffers, Calculation, and so on.
 */
void main(void)
{

    if(sizeof(BUF)!=BUFSIZE) {
        HandleError(BUFFER_SIZE_WRONG);
    }

    InitCpu();            // init PMST and bank switch regs

    InitBuffers();

    InitAuxport();

    InitDmaport();

    InitCodec();

    InitDma();

    InitError();          // initialize error processing

    InitVoice();
    InitFilter();
    InitPestr();
    InitTone();

    InitCalc();           // initialize the calc object

    InitProcess();        // initialize the procman object

    StartAuxport();      // get Auxport object going

    StartDma();          // enable and unmask DMA's

    StartDmaport();      // enable McBSP2

    EnableInterrupts();

    for(;;) {           // processing loop
        Process();
    }
}

/*
 * auxport.h
 *
 *     Header file for the Auxport object
 *
 */

#ifndef AUXPORT_H
#define AUXPORT_H
/*
 * a bit more than the pot value seen
 * when the potentiometer is turned full up
 * This value was set by observing the
 * potentiometer value with the debugger.
 */
#define MAX_POTVAL (0xC00)
/*
 * public methods
 */
void InitAuxport (void);
void StartAuxport (void);
bool SinePressed (void);
bool TrianglePressed (void);
bool SawPressed(void);
bool SquarePressed(void);
bool PotChanged(void);
u16 LatestPot(void);
#endif

/*
```

```
 * auxport.c
 *
 *    A voice substitution device module.
 *    Copyright 2003, David Clifton
 *    All Rights Reserved
 *
 *    data & methods for auxport object
 *
 */
#include "types.h"
#include "cpu.h"
#include "error.h"
#include "auxport.h"
/*
 * McBSP0 serial port address and register sub-addresses
 */
#define DRR2 ( * (volatile u16 *) 0x0020) // MCBSP0 Data Receive Reg 2
#define DRR1 ( * (volatile u16 *) 0x0021) // MCBSP0 Data Receive Reg 1
#define DXR2 ( * (volatile u16 *) 0x0022)
#define DXR1 ( * (volatile u16 *) 0x0023)

#define MCBSP0A ( * (volatile u16 *) 0x0038) // McBsp0 Subaddress Register
#define MCBSP0D ( * (volatile u16 *) 0x0039) // McBsp0 Data Registers
#define SPCR1  (0x0000)    // SPCR1 sub-address
#define SPCR2  (0x0001)    // SPCR2 sub-address
#define RCR1   (0x0002)    // RCR1 sub-address
#define RCR2   (0x0003)    // RCR2 sub-address
#define SRGR1  (0x0006)    // SRGR1 sub-address
#define SRGR2  (0x0007)    // SRGR2 sub-address
#define PCR    (0x000E)    // PCR sub-address
/*
 * SPCR1 bit field settings
 *
 * 00011RR00000000
 * ||||||||||||||+-- RRST Receiver in Reset (initially)
 * |||||||||||||+--- RRDY Receiver ready status bit
 * ||||||||||||+---- RFULL Receiver overrun status bit
 * |||||||||||+----- RSYNCERR Dont want to know
 * |||||||||++------ RINTM  Interrupt on end of word
 * ||||||||+-------- ABIS  abis mode disabled
 * |||||||+--------- DXENA DX mode disabled
 * |||||++---------- Reserved
 * |||++------------ CLKSTP Clock starts rising edge, with delay
 * |++-------------- RJUST Right Justify, zero fill MSBs
 * +---------------- DLB Digital Loopback off
 */
#define INIT_SPCR1 (0x1800)
#define SPCR1_RXENABLE  (0x1801) // Put this into SPCR1 when ready to receive
/*
 * SPCR2 bit field settings
 *
 * RRRRRR0000000000
 * ||||||||||||||+-- XRST  Transmitter in Reset (permanently)
 * |||||||||||||+--- XRDY  Transmitter ready status bit
 * ||||||||||||+---- XEMPTY Transmitter empty status bit
 * |||||||||||+----- XSYNCERR Transmitter sync eror bit
 * ||||||||++------ XINTM  Transmit interrupt mode (end of word)
 * ||||||||+-------- GRST Sample rate generator is reset
 * |||||||+--------- FRST Frame synch generator is reset
 * |||||||+---------- SOFT Stop serial port immediately on breakpoint
 * ||||||+----------- FREE Don't allow transmit to run on breakpoint
 * RRRRRR------------ Reserved
 */
#define INIT_SPCR2 (0x0000)
#define ENABLE_SRG (0x0040) // start the sample rate generator
/*
 * PCR bit field settings
 *
 * RR000000RXXX1101
 * ||||||||||||||+-- CLKRP Sample on rising edge of  CLKR
 * |||||||||||||+--- CLKXP Transmit clock polarity (low inactive)
 * ||||||||||||+---- FSRP Receive Frame Synch (active low)
 * |||||||||||+----- FSXP  Transmit Frame Synch Polarity (active low)
 * ||||||||+++------ Receive Port Pin Values (irrelevant)
 * |||||||+--------- Reserved
 * ||||||+---------- CLKRM Receive Clock Mode (hooked up to CLKX)
 * |||||+----------- CLKXM Transmit Clock Mode (Input)
 * ||||+------------ FSRM  Receive Frame Synch (generated from FSXM)
 * |||+------------- FSXM  Transmit Frame Synch (Input)
 * ||+-------------- RIOEN Receive port pins enabled as serial port
 * ++--------------- XIOEN Transmit port pins enabled as serial port
 * ++--------------- Reserved
 */
#define INIT_PCR (0x000D)
/*
 * RCR1 bit field settings
 *
 * R0000000010RRRRR
 * ||||||||||+++++---- Reserved
 * |||||||+++--------- RWDLEN1 Word length is 16 bits
 * |+++++++------------ RFRLEN1 Frame length is 1 word
 * +--------------------Reserved
 */
```

```
#define INIT_RCR1 (0x0040)
/*
 * RCR2 bit field settings
 *
 * 0000000000000000
 * ||||||||||||||++---- RDATDLY No receive data deley
 * ||||||||||||||+------ RFIG Frame sync pulses after first restart xfer
 * ||||||||||||++------- RCOMPAND No companding
 * ||||||||+++---------- RWDLEN2 Receive word length 2 is 8 bits (dont care)
 * |+++++++------------- RFRLEN2 Receive frame length 2 is 1 word (dont care)
 * +--------------------RPHASE Single phase frame
 */
#define INIT_RCR2 (0x0000)
/*
 * SRGR1 bit field settings
 *
 * 0000000000000001
 * ||||||||++++++++---- CLKGDV  sample rate generator divide #=2
 * ++++++++------------ FWID  Frame width (set by other end)
 */
#define INIT_SRGR1 (0x0001)
/*
 * SRGR2 bit field settings
 *
 * 0010000000000000
 * ||||++++++++++++---- FPER  Frame Period (Ignored)
 * |||+----------------- FSGM  Frame Sync mode (copy)
 * ||+------------------ CLKSM Derive sample rate clock from CPU clock
 * |+------------------- CLKSP Not used
 * +-------------------- GSYNC Not used
 */
#define INIT_SRGR2 (0x2000)

/* ---------------------------------------------------------------
 * Auxport object data
 * --------------------------------------------------------------- */
bool sinePressed=FALSE;
bool trianglePressed= FALSE;
bool sawPressed=FALSE;
bool squarePressed= FALSE;
bool potChanged=FALSE;
u16 prevPot,currPot;

/* ---------------------------------------------------------------
 * Auxport public methods
 * --------------------------------------------------------------- */
/*
 * InitAuxport
 *
 *     This routine sets up McBSP0 for use as a SPI slave
 * to the AuxIn object on the basic stamp 2 board.
 */
void InitAuxport(void)
{
  /*
   * Initialize McBsp0
   */
   MCBSP0A=SPCR1;
   MCBSP0D=INIT_SPCR1;   // Reset the McBSP0 transmitter
   MCBSP0A=SPCR2;
   MCBSP0D=INIT_SPCR2;   // Reset the McBSP0 receiver
   MCBSP0A=PCR;
   MCBSP0D=INIT_PCR;
   MCBSP0A=RCR1;
   MCBSP0D=INIT_RCR1;
   MCBSP0A=RCR2;
   MCBSP0D=INIT_RCR2;
   MCBSP0A=SRGR1;
   MCBSP0D=INIT_SRGR1;
   MCBSP0A=SRGR2;
   MCBSP0D=INIT_SRGR2;
   currPot=0;
   prevPot=0;
   sinePressed=FALSE;
   trianglePressed=FALSE;
   sawPressed=FALSE;
   squarePressed=FALSE;
   potChanged=FALSE;
}
/*
 * StartAuxport
 *
 *    Start up McBSP0 as the SPI slave of the AuxIn object on
 * the basic stamp 2 board.  Call this when you are sure that
 * interrupts will be enabled within the next millisecond or so.
 */
void StartAuxport(void)
{
   MCBSP0A=SPCR2;      // select SPCR2 register
   MCBSP0D=ENABLE_SRG; // start sample rate generator

/*
 * kill a few hundred microseconds
```

```
 */
   pause(1000);       // have to check this with scope

/*
 * enable the receiver and transmitter
 */
   MCBSP0A=SPCR1;     // select the SPCR1 register
   MCBSP0D = SPCR1_RXENABLE;
   MCBSP0A=SPCR2;     // select the SPCR2 register
   MCBSP0D = ENABLE_SRG; // enable the sample rate generator
/*
 * unmask the receive interrupt
 */
   *IFR |=0x10;
   *IMR |=0x10;
}
/*
 * SinePressed
 *
 *   This routine returns the value of the sinePressed flag,
 * and sets the flag to FALSE.
 */
bool SinePressed(void)
{
    bool retval;

    retval=sinePressed;
    sinePressed=FALSE;
    return retval;
}
/*
 * TrianglePressed
 *
 *   This routine returns the value of the trianglePressed flag,
 * and sets the flag to FALSE.
 */
bool TrianglePressed(void)
{
    bool retval;

    retval=trianglePressed;
    trianglePressed=FALSE;
    return retval;
}
/*
 * SawPressed
 *
 *   This routine returns the value of the sawPressed flag,
 * and sets the flag to FALSE.
 */
bool SawPressed(void)
{
    bool retval;

    retval=sawPressed;
    sawPressed=FALSE;
    return retval;
}
/*
 * SquarePressed
 *
 *   This routine returns the value of the squarePressed flag,
 * and sets the flag to FALSE.
 */
bool SquarePressed(void)
{
    bool retval;

    retval=squarePressed;
    squarePressed=FALSE;
    return retval;
}

/*
 * PotChanged
 *
 *   This routine returns the value of the potChanged flag,
 * and sets the flag to FALSE.
 */
bool PotChanged(void)
{
    bool retval;

    retval=potChanged;
    potChanged=FALSE;
    return retval;
}

/*
 * LatestPot
 *
 *    This routine returns the current value of the
 * potentiometer setting.
```

```
 */
u16 LatestPot(void)
{
    return currPot;
}
/* ------------------------------------------------------------
 * Auxport private methods
 * ------------------------------------------------------------ */
/*
 * AuxportRxInterrupt
 *
 *   This routine receives the next serial word from
 * the Basic Stamp 2 board, via McBsp0.  If either of
 * the buttons were pressed, sets plusPressed and/or
 * minusPressed appropriately.  Sets latestPot to the
 * low order six bits of the word.
 */
interrupt void AuxportRxInterrupt(void)
{
    u16 rxword1;

    rxword1=DRR1;
/*
 * get current pot setting
 * set flag if it has changed
 */
    prevPot=currPot;
    currPot= (rxword1 & 0xFFF);
    if(currPot/16!=prevPot/16) potChanged=(1==1);
/*
 * set flags for buttons pressed
 */
    if((rxword1 & 0x8000)!=0) sinePressed=(1==1);
    if((rxword1 & 0x4000)!=0) trianglePressed=(1==1);
    if((rxword1 & 0x2000)!=0) sawPressed=(1==1);
    if((rxword1 & 0x1000)!=0) squarePressed=(1==1);
}

/*
 * bufman.h
 *
 *     Header file for bufman object
 *
 */
#ifndef BUFMAN_H
#define BUFMAN_H
/*
 * frame and sample rates
 */
#define FRAME_RATE (200)        // input is 200 buffers/second
#define SAMPLE_RATE (48000)     // sample rate is 48000 samples/second
/*
 * number of left channel samples in a buffer
 */
#define BUFSIZE (500)
#define LEFTCHAN_SAMPLES (SAMPLE_RATE/FRAME_RATE)
/*
 * type definition for BUF
 */
typedef struct {
    s32 bufcount;               // buffer sequency number set on allocation
    s16 nextBuf;                // bufArray index of next buffer in chain
    s16 data[2*LEFTCHAN_SAMPLES]; // array of left channel samples
    s16 filler[BUFSIZE - 2*LEFTCHAN_SAMPLES - 3];  // have buffer size power of 2 words
                                //  ( could simplify linker issues )
} BUF;
/*
 * external def of buffer array
 * used only before dma startup
 * to initialize src and destination
 * registers
 */
extern BUF buffers[];
/*
 * Public method definitions
 */
void InitBuffers(void);
void PutBuf(void);
BUF *AllocBuf(void);
BUF *ReceiveBuf(void);
void OutputBuf(void);
BUF *GetBuf(void);
void FreeBuf(void);
#endif

/*
 * bufman.c
 *
 *    A voice substitution device module.
 *    Copyright 2003, David Clifton
 *    All Rights Reserved
 *
 *    data & methods for buffer management object
```

```
 *
 */
#include "types.h"
#include "cpu.h"
#include "error.h"
#include "bufman.h"

/* ------------------------------------------------------------------
 * Buffer object data
 * ------------------------------------------------------------------ */
#define NUM_BUFFERS 5
BUF *bufptrs[NUM_BUFFERS]; // pointers to buffers
BUF buffers[NUM_BUFFERS];  // buffers for data
s32 totCount=0;    // incremented with every buffer allocation
s16 firstFree=NIL;   // index of first free buffer
s16 inBuf=NIL;     // index of buffer receiving new data via DMA
s16 firstProcess=NIL; // index of first buffer in process list
s16 calcBuf=NIL;  // index of buffer on which calculations are performed
s16 firstOut=NIL;  // index of first buffer in output list
s16 outBuf=NIL;       // index of buffer being output via DMA

/*------------------------------------------------------------------
 * Public method definitions
 * ------------------------------------------------------------------ */
/*
 * InitBuffers
 *   Initialize all the buffers and place them on
 * the free buffer list.
 */
void InitBuffers(void)
{
    s16 j,k;

    totCount=0;
    firstFree=0;
    inBuf=NIL;
    firstProcess=NIL;
    calcBuf=NIL;
    firstOut=NIL;
    outBuf=NIL;
    for(j=0;j<NUM_BUFFERS;j++) {
        buffers[j].bufcount=0;  // set by alloc routine
        buffers[j].nextBuf=j+1;
        for(k=0;k<(2*LEFTCHAN_SAMPLES);k++) {
            buffers[j].data[k]=0;
        }
        for(k=0;k<(BUFSIZE - 2*LEFTCHAN_SAMPLES - 3);k++) {
            buffers[j].filler[k]=0xDEAD;
        }
        bufptrs[j]=&buffers[j];
    }
    buffers[NUM_BUFFERS-1].nextBuf=NIL;
}
/*
 * AllocBuf
 *
 * If inBuf is not NIL, call the error handler.
 * If there are no buffers on the free list,
 * call the error handler.
 * Otherwise, get a new buffer from the free chain
 * and put its index into inBuf.  Return a pointer
 * to it.
 *
 * NOTE:
 *    This routine will always be called from the
 * DMAIn() interrupt routine.
 */
BUF *AllocBuf(void)
{
      BUF *dabuf;


      if(inBuf!=NIL) {
          HandleError(INBUF_BUSY);
          dabuf=NULL;
      } else if(firstFree==NIL) {
          HandleError(OUT_OF_BUFFERS);
          dabuf=NULL;
      } else {
          inBuf=firstFree;
          dabuf=&buffers[inBuf];
          firstFree=dabuf->nextBuf;
          dabuf->nextBuf=NIL;
          dabuf->bufcount=totCount++;
      }
      return dabuf;
}
/*
 * PutBuf
 *
 *    If there is no buffer indicated by inBuf,
 * this routine calls the error handler.  If there
 * is a buffer indicated by inBuf, this routine adds
```

```
 * it to the end of the process chain, and sets
 * inbuf to NIL.
 *
 * This routine is always called from the
 * DMAIn() interrupt routine.
 */
void PutBuf(void)
{
    s16 next,prev;

    if(inBuf==NIL) {
        HandleError(MISSING_INBUF);
    } else if(firstProcess==NIL) {
        firstProcess=inBuf;
    } else {
        prev=NIL;
        next=firstProcess;
        while(next!=NIL) {
            prev=next;
            next=buffers[prev].nextBuf;
        }
        buffers[prev].nextBuf=inBuf;
    }
    buffers[inBuf].nextBuf=NIL;

    inBuf=NIL;
}
/*
 * ReceiveBuf
 *
 *   If calcBuf is not NIL, calls the error handler.
 *   If there is no buffer in the process list, returns
 *   NULL to the caller.
 *   If there is a buffer in the process list, removes
 *   the first such buffer, places its index into calcBuf,
 *   and returns a pointer to the buffer.
 *
 *   This routine is called in the background, and so must
 *   disable interrupts while removing the buffer from the
 *   process list, in case the DMAIn() interrupt should happen
 *   while it is doing it.
 */
BUF *ReceiveBuf(void)
{
    BUF *dabuf;

    DisableInterrupts();
    if(calcBuf!=NIL) {
        HandleError(CALC_IN_PROGRESS); // don't receive until calc finished
        dabuf=NULL;      // wait for calc position to be free
    } else if(firstProcess==NIL) {
        dabuf=NULL;
    } else {
        calcBuf=firstProcess;
        dabuf=&buffers[calcBuf];
        firstProcess=dabuf->nextBuf;
        dabuf->nextBuf=NIL;

    }

    EnableInterrupts();
    return dabuf;
}
/*
 * OutputBuf
 *
 *   If calcBuf is NIL, this routine calls the
 * error handler.  Otherwise, it puts the buffer
 * indicated by calcBuf onto the output buffer
 * list, and sets calcBuf to NIL.
 *
 * This routine is called in the background, and so must
 * disable interrupts while adding the buffer to the
 * output list, in case the DMAOut() interrupt should happen
 * while it is doing it.
 */
void OutputBuf(void)
{
    s16 prev,next;

    DisableInterrupts();
    if(calcBuf==NIL) {
        HandleError(NO_BUF_TO_OUTPUT);
    } else if(firstOut==NIL) {
        firstOut=calcBuf;
        calcBuf=NIL;
        buffers[firstOut].nextBuf=NIL;
    } else {
       prev=NIL;
        next=firstOut;
        while(next!=NIL) {
            prev=next;
            next=buffers[prev].nextBuf;
```

```
        }
        buffers[prev].nextBuf=calcBuf;
        buffers[calcBuf].nextBuf=NIL;
        calcBuf=NIL;
    }

    EnableInterrupts();
}
/*
 * FreeBuf
 *
 *   If outBuf is NIL, call the error handler.
 *   Otherwise put the buffer indicated by outBuf
 *   onto the free chain.  Set outBuf to NIL.
 *
 *   This method is only called from within the
 *   DMAOut() interrupt routine, and so there is
 *   no need to disable interrupts.
 */
void FreeBuf(void)
{
    if(outBuf==NIL) {
        HandleError(NO_BUF_TO_FREE);
    } else {
        buffers[outBuf].nextBuf=firstFree;
        firstFree=outBuf;
        outBuf=NIL;
    }
}
/*
 * GetBuf
 *
 *    If outBuf is not NIL, call the error handler.
 *    If there are no output list buffers, call the error handler.
 *    Otherwise, remove the first buffer from the output list
 *    and place it in the outBuf station. Return a pointer
 *    to the first word of the free buffer.
 *
 *   This method is only called from within the
 *   DMAOut() interrupt routine, and so there is
 *   no need to disable interrupts.
 */
 BUF *GetBuf(void)
 {
     BUF *dabuf=NULL;

     if(outBuf!=NIL) {
         HandleError(STILL_USING_OUTPUT_BUF);
     } else if(firstOut==NIL) {
         HandleError(NO_OUTPUT_BUFFERS);
     } else {
         outBuf=firstOut;
         dabuf=&buffers[outBuf];
         firstOut=dabuf->nextBuf;
     }

     return dabuf;
 }

/*
 * calc.h
 *
 *     Header file for the Calc object
 *
 */
#ifndef CALC_H
#define CALC_H
/*
 * public methods
 */
void InitCalc (void);
s16 Calc(BUF *bptr);
#endif

/*
 * calc.c
 *
 *    A voice substitution device module.
 *    Copyright 2003, David Clifton
 *    All Rights Reserved
 *
 *  This module manages the signal processing
 * calculations for the voice substitution device.
 *
 */
#include "types.h"
#include "error.h"
#include "bufman.h"
#include "voice.h"
#include "tone.h"
#include "calc.h"
/*
 * calc states
```

```
 */
typedef enum { CSTART=NIL,CSTATE0=0,CSTATE1,CSTATE2,
               CSTATE3,CSTATE4 } CSTATE;
/*
 * calc data
 */
CSTATE cstate=CSTART;     // current calculation state
BUF *bptr=NULL;           // buffer pointer
u32 avgAmplitude=0;       // avg amplitude in buffer supplied
s16 pitchFreq=0;          // pitch frequency or NIL if unvoiced

/* ------------------------------------------------
 * public methods
 * ------------------------------------------------ */
/*
 * InitCalc
 *    Initialize the calc state and all of the
 * algorithms employed by the calc state machine.
 */
void InitCalc(void)
{
    cstate=CSTATE0;
    bptr=NULL;
//    InitVoice();
//    InitTone();
}
/*
 * Calc
 *    Execute the calculation state machine
 */
s16 Calc(BUF *ptr)
{

    bptr=ptr;
    switch(cstate) {
    case CSTART:
        break;

    case CSTATE0:
        avgAmplitude=(3*avgAmplitude+CopyBuffer(ptr))/4;
        cstate=CSTATE1;
        break;

    case CSTATE1:
        FilterBuffer();
        cstate=CSTATE2;
        break;

    case CSTATE2:
        pitchFreq=EstimatePitch();
        cstate=CSTATE3;
        break;

    case CSTATE3:
        if(pitchFreq!=NIL) {
            cstate=CSTATE4;
        } else {
            cstate=CSTATE0;
        }
        break;

    case CSTATE4:
        ToneInject(bptr,(u16)avgAmplitude,pitchFreq);
        cstate=CSTATE0;
        break;
    }
    return (s16)cstate;
}

/*
 * codec.h
 *
 *     Header file for the Codec object
 *
 */

#ifndef CODEC_H
#define CODEC_H
/*
 * codec sample rate
 */
#define CODEC_SAMPLE_RATE (48000)
/*
 * public methods
 */
void InitCodec (void);
void SetVolume(u16 vol);
#endif

/*
 * codec.c
 *
 *    A voice substitution device module.
```

```
 *    Copyright 2003, David Clifton
 *    All Rights Reserved
 *
 *    data & methods for codec object
 *    which initializes and sets volume
 *    for the the on-board
 *    Burr-Brown PCM3002 codec.
 *
 */
#include "types.h"
#include "auxport.h"
#include "codec.h"
/*
 * Initialization values for the four
 * codec registers
 * ( including register designators in bits 9 and 10 )
 */
#define CREG0INITH 0x01  // left chan DAC atten 0dB
#define CREG0INITL 0xFF
#define CREG1INITH 0x03  // right chan DAC atten 0dB
#define CREG1INITL 0xFF
#define CREG2INITH 0x04  // power down modes off
#define CREG2INITL 0x00
#define CREG3INITH 0x06  // Use Format 0 bit streams
#define CREG3INITL 0x00
/*
 * Mask for Codec Ready bit in CPLD MISC register
 */
#define CODEC_NRDY 0x80
/*
 * I/O space Addresses of CPLD supplied access to codec
 * registers on the 5416DSK board.
 */
ioport unsigned port2;          // CPLD CODEC_L_CMD Register
ioport unsigned port3;          // CPLD CODEC_H_CMD Register
ioport unsigned port6;          // CPLD MISC Register

/*
 * InitCodec
 *
 *    This routine sets up the codec four codec
 * registers by writing them through the CPLD's
 * low and high command registers.
 */
void InitCodec(void)
{

    while((port6 & CODEC_NRDY)!=0)
         ;    // wait for codec ready

    port2=CREG0INITL;
    port3=CREG0INITH;

    while((port6 & CODEC_NRDY)!=0)
         ;    // wait for codec ready

    port2=CREG1INITL;
    port3=CREG1INITH;

    while((port6 & CODEC_NRDY)!=0)
         ;    // wait for codec ready

    port2=CREG2INITL;
    port3=CREG2INITH;

    while((port6 & CODEC_NRDY)!=0)
         ;    // wait for codec ready

    port2=CREG3INITL;
    port3=CREG3INITH;

    while((port6 & CODEC_NRDY)!=0)
         ;    // wait for codec ready

}

/*
 * SetVolume
 *    Set the codec output attenuation according to
 * the value supplied (0-255).
 */
void SetVolume(u16 vol)
{

    while((port6 & CODEC_NRDY)!=0)
         ;    // wait for codec ready

    port2=vol;
    port3=CREG0INITH;

}

/*
```

```
 * cpu.h
 *
 *     Header file for the Cpu object
 *
 */

#ifndef CPU_H
#define CPU_H

/*
 * interrupt registers
 */
extern volatile u16 *IMR;
extern volatile u16 *IFR;
/*
 * public methods
 */
void InitCpu (void);
void DisableInterrupts(void);
void EnableInterrupts(void);
void pause(s16);
#endif

/*
 * cpu.c
 *
 *    A voice substitution device module.
 *    Copyright 2003, David Clifton
 *    All Rights Reserved
 *
 * This module initializes the  TI 5416 DSP chip,
 * including the PSTM register bits OVLY, MP/~MC,
 * and the interrupt vector locations, the clock
 * mode retister, CLKMD, the bank switch register,
 * BSCR.
 * It also provides routines to disable and enable
 * hardware interrupts.
 *
 */

#include "types.h"
#include "error.h"
#include "cpu.h"
/*
 * two settings for CLKMD register
 */
#define PLL_DIV_MODE (0x0000)
#define CLKMD_FOR_80MHZ (0x4106)
#define CLKMD_FOR_160MHZ (0x9146)
/*
 * CLKMD status bit
 */
#define CLKMD_STATUS (1)
/*
 * interrupt related registers
 */
volatile u16 *IMR = (u16 *) 0x0;    // interrupt mask register ptr
volatile u16 *IFR = (u16 *) 0x1;    // interrupt flag register ptr
volatile u16 *ST0 = (u16 *) 0x6;    // status register 0 ptr
volatile u16 *ST1 = (u16 *) 0x7;    // status register 1 ptr
/*
 * bank switch control register
 */
volatile u16 *BSCR = (u16 *)0x29;   // bank switch control register ptr
/*
 * clock mode register
 */
volatile u16 *CLKMD = (u16 *)0x58;  // clock mode register

/* ------------------------------------------------------------
 * Private method declarations
 * ------------------------------------------------------------ */
void pause(s16);


/* ------------------------------------------------------------
 * Public methods of Cpu object
 * ------------------------------------------------------------ */
/*
 * InitCpu
 *      This routine sets up the processor's
 * PMST register, its bankswitch control
 * register, and the phase locked loop
 * which generates the system clock.
 *
 */
void InitCpu (void)
{
  *IMR=0;     // clear interrupt mask register
  *IFR=0;     // and interrupt flag register
/*
 * initialize pocessor mode status register
 * (OVLY, MP/~MC, and interrupt vector location)
```

```
 */
  asm (" STM #7FF0h,1Dh");
/*
 * initialize bank switch register
 */
  *BSCR = 0x0002;
  asm ("    NOP");
  asm ("    NOP");
  asm ("    NOP");
/*
 * Set up phase locked loop for
 * 160 MHz system clock
 *
 * On the 5416_DSK board from Spectrum Digital:
 *
 * The external clock is 16MHz
 *
 * Use jumpers to set CLKMD1,CLKMD2, and CLKMD3 pins
 * to 1,0,1. This will cause the chip to powerup with
 * the PLL locked to clock the frequency of
 * 16MHz.
 *
 * SETTING THE CLKMD REGISTER
 * The CLKMD register has the following subfields:
 *
 * MMMMDCCCCCCCCONS
 * ||||||||||||||+------ PLLSTATUS
 * |||||||||||||+------- PLLNDIV
 * ||||||||||||+-------- PLLON/OFF
 * |||||++++++++--------- PLLCOUNT
 * ||||+------------------ PLLDIV
 * ++++------------------- PLLMUL
 *
 * This routine will set up the PLL to produce 160MHz.
 * It does so by setting PLLNDIV to 1, PLLDIV to 0,
 * and PLLMUL to 9.  The PLLCOUNT field in the CLKMD
 * register is set according to the computation
 * below:
 *
 *   PLLCOUNT > (Lockup time in usec)*(external clock freq in MHz)/16
 *
 * According to Figure 8-5 of the 54x CPU & Peripherals volume,
 * The lockup time for an 112MHz clock frequency must be at least
 * 40usec.
 *
 *   PLLCOUNT = 40*16/16 =40
 *
 * Plugging all these values in gives a CLKMD
 * register value of:
 *
 *   0100 0 00101000 1 1 0
 *        or
 *   CLKMD <-- 0x9146
 */

    *CLKMD=PLL_DIV_MODE;  // set up CLKMD register for DIV mode

    while((*CLKMD & CLKMD_STATUS)!=0)
    {
        ;               // wait to take effect
    }

    *CLKMD =  CLKMD_FOR_160MHZ;   // set for 160 MHZ

    pause(100);         // wait a bit
}


/*
 * DisableInterrupts(void);
 *
 *    Disable interrupts unconditionally
 */
void DisableInterrupts(void)
{
    /*
     * physically disable interrupts
     */
    asm ("   intr   2 ");    // interrupt vector returns with ret
                             // not rete

}
/*
 * EnableInterrupts
 *
 *   unconditionally enable interrupts
 */
void EnableInterrupts(void)
{
    /*
     * physically enable interrupts
     */
      asm ("  rsbx    1,INTM ");    // enable interrupts
```

```
}

/*
 * pause
 *
 *    Wait for a time period that varies geometrically
 * with the single argument.
 */
void pause(s16 val)
{
    s16 i,j;

    j=val;
    while (j>0) {
        for(i=1;i<j--;i++)
            ;
    }

}
/* ------------------------------------------------
 * Local routines
 * ------------------------------------------------ */
/*
 * SpuriousInterrupt
 *
 * Spurious hardware interrupt handler
 */
void interrupt SpuriousInterrupt (void)
{
  HandleError (SPURIOUS_INTERRUPT);
}
/*
 * SpuriousTrap
 *
 * Spurious trap handler
 */
void interrupt SpuriousTrap (void)
{
  HandleError (SPURIOUS_TRAP);
}

/*
 * dma.h
 *
 *  Header file for the DMA object
 *  Which manages DMA channels 4 and 5
 *  for output and input respectively.
 *
 */

#ifndef DMA_H
#define DMA_H
/*
 * public methods
 */
void InitDma (void);
void StartDma (void);
#endif

/*
 * dma.c
 *
 *    A voice substitution device module.
 *    Copyright 2003, David Clifton
 *    All Rights Reserved
 *
 *    data & methods for dma object
 *    which connects the McBsp2 input channel
 *    to DMA channel 5, and the McBsp2 output
 *    channel to DMA channel 4.
 *
 */
#include "types.h"
#include "cpu.h"
#include "error.h"
#include "bufman.h"
#include "dmaport.h"
#include "dma.h"
/*
 * DMA register addresses and sub-addresses
 */
#define DMPREC ( * (volatile u16 *) 0x0054) // Channel priority and Enable Control Register
#define DMSA ( * (volatile u16 *) 0x0055) // Subbank Address Register
#define DMSDI ( * (volatile u16 *) 0x0056) // Subbank Data Access w/ address postincrement
#define DMSDN ( * (volatile u16 *) 0x0057) // Subbank Data Access w/ address postdecrement
/*
 * DMA channel 4 sub-addresses
 */
#define DMSRC4  (0x0014)   // Source address register
#define DMDST4  (0x0015)   // Dest address register
#define DMCTR4  (0x0016)   // Element count register
#define DMSFC4  (0x0017)   // Sync select and frame count register
#define DMMCR4  (0x0018)   // Transfer Mode Control register
```

```
/*
 * DMA channel 5 sub-addresses
 */
#define DMSRC5  (0x0019)   // Source address register
#define DMDST5  (0x001A)   // Dest address register
#define DMCTR5  (0x001B)   // Element count register
#define DMSFC5  (0x001C)   // Sync select and frame count register
#define DMMCR5  (0x001D)   // Transfer Mode Control register
/*
 * Other DMA sub-addresses
 */
#define DMASRCP  (0x001E)   //  Source Program Page Address (all channels)
#define DMADSTP  (0x001F)   //  Destination Program Page Address (all channels)
#define DMIDX0   (0x0020)   //  Element Address Index Register 0
#define DMIDX1   (0x0021)   //  Element Address Index Register 1
#define DMFRI0   (0x0022)   //  Frame Address Index Register 0
#define DMFRI1   (0x0023)   //  Frame Address Index Register 1
#define DMGSA    (0x0024)   //  Global source address reload register
#define DMGDA    (0x0025)   //  Global destination address reload register
#define DMGCR    (0x0026)   //  Global element count reload register
#define DMGFR    (0x0027)   //  Global frame count reload register
/*
 *
 * Set up DMA Channels 4 and 5 as output and input
 * channels respectively.
 *
 * DMPREC bit field settings (Priority and Enable Control)
 *
 * 1R00000000000000
 * ||||||||||||||+- DE[0] Channel 0 enable bit
 * |||||||||||||+-- DE[1] Channel 1 enable bit
 * ||||||||||||+--- DE[2] Channel 2 enable bit
 * |||||||||||+---- DE[3] Channel 3 enable bit
 * ||||||||||+----- DE[4] Channel 4 enable bit
 * |||||||||+------ DE[5] Channel 5 enable bit
 * |||||||++-------- INTOSEL Chan4->Intrpt#12 Chan5->intrpt#13
 * |||||||+--------- DPRC[0] Channel 0 is lower priority
 * ||||||+---------- DPRC[1] Channel 1 is lower priority
 * |||||+----------- DPRC[2] Channel 2 is lower priority
 * ||||+------------ DPEC[3] Channel 3 is lower priority
 * |||+------------- DPRC[4] Channel 4 is lower priority
 * ||+-------------- DPRC[5] Channel 5 is lower priority
 * |+--------------- Reserved
 * +---------------- FREE DMA transfer stops when emulation stops
 */
#define INIT_DMPREC (0x8000)
#define DMPREC_ENABLE45  (0x0030) // Put this into DMPREC when ready to roll
#define DMPREC_ENABLE4   (0x0010) // Or this into DMPREC to enable Ch 4
#define DMPREC_ENABLE5   (0x0020) // Or this into DMPREC to enable Ch 5
/*
 * Channel 5 (Input Channel) Register Settings
 *
 * DMSRC5 - Channel 5 Source Address Register
 *
 * AAAAAAAAAAAAAAAA
 * ++++++++++++++++-- low order sixteen bits of extended address
 *                    of source address for input channel.  This
 *                    should be the address of the McBSP2 receive
 *                    register DRR1 (see dmaport.h)
 */
#define INIT_DMSRC5 ((u16)0x0031);  // address of DRR1
/*
 * DMDST5 - Channel 5 Destination Address Register
 *
 * AAAAAAAAAAAAAAAA
 * ++++++++++++++++-- low order sixteen bits of extended address
 *                    of destination address for input channel.  This
 *                    should be the address of the buffer allocated by
 *                    bufman during the channel start and DMAIn interrupt
 *                    routines.  For initialization just use the address
 *                    of the first buffer in the buffers array in bufman.c.
 */
#define INIT_DMDST5 ((u16)buffers);
/*
 * DMCTR5 - Channel 5 Element Count Register
 *
 * CCCCCCCCCCCCCCCC
 * ++++++++++++++++-- Elements to move per frame.  This should be one
 *                    less than the actual frame size.
 */
#define INIT_DMCTR5 ((u16)(2*LEFTCHAN_SAMPLES-1))
/*
 * DMSFC5 -- Channel 5 Frame Sync Register
 *
 * 00110RRR00000000
 * ||||||||++++++++-- Frame Count 0 means one frame
 * |||||+++---------- Reserved
 * ||||+------------- Double word mode 0 means single word
 * ++++-------------- Synch event 0011 is the McBsp2 Receive event
 */
#define INIT_DMSFC5 ((u16)0x3000)
/*
 * DMMCR5 -- Channel 5 Transfer mode register
```

```
 *
 * 0100R000 01R001 01
 * |||||||| |||||| ++--- DMD destination address space is DATA
 * |||||||| |||+++ ----- DIND dest address transfer index mode (++)
 * |||||||| ||+--- ----- Reserved
 * |||||||| ++---- ----- DMS source address space is DATA
 * |||||+++ ------ ----- SIND src address transfer index mode (no mod)
 * ||||+--- ------ ----- Reserved
 * |||+---- ------ ----- CTMOD Multiframe mode (not ABU)
 * ||+----- ------ ----- IMOD Interrupt at end of block
 * |+------ ------ ----- DINM Interrupt based on IMOD bit
 * +------- ------ ----- Autoinitialization disabled
 */
#define INIT_DMMCR5 ((u16)0x4045)

/*
 * Channel 4 (Output Channel) Register Settings
 *
 * DMSRC4 - Channel 4 Source Address Register
 *
 * AAAAAAAAAAAAAAAA
 * ++++++++++++++++-- low order sixteen bits of extended address
 *                   of source address for input channel.   This
 *                   should be the address of the buffer allocated by
 *                   bufman during the channel start and DMAOut interrupt
 *                   routines.  For initializaiton just use the address
 *                   of the second buffer in the buffers array in bufman.c.
 */
#define INIT_DMSRC4 ((u16)(buffers+sizeof(BUF)));
/*
 * DMDST4 - Channel 4 Destination Address Register
 *
 * AAAAAAAAAAAAAAAA
 * ++++++++++++++++-- low order sixteen bits of extended address
 *                   of destination address for input channel.  This
 *                   should be the address of the McBSP2 transmit
 *                   register DXR1 (see dmaport.h)
 */
#define INIT_DMDST4 ((u16)0x0033);  // Address of DXR2
/*
 * DMCTR4 - Channel 4 Element Count Register
 *
 * CCCCCCCCCCCCCCCC
 * ++++++++++++++++-- Elements to move per frame.  This should be one
 *                   less than the actual frame size.
 */
#define INIT_DMCTR4 ((u16)(2*LEFTCHAN_SAMPLES-1))
/*
 * DMSFC4 -- Channel 4 Frame Sync Register
 *
 * 01000RRR00000000
 * ||||||||++++++++-- Frame Count 0 means one frame
 * |||||+++---------- Reserved
 * ||||+------------- Double word mode 0 means single word
 * ++++-------------- Synch event 0100 is the McBsp2 Transmit event
 */
#define INIT_DMSFC4 ((u16)0x4000)
/*
 * DMMCR4 -- Channel 4 Transfer mode register
 *
 * 0100R00101R00001
 * ||||||||||||||++--- DMD destination address space is DATA
 * ||||||||||||+++----- DIND dest address transfer index mode (no mod)
 * ||||||||||+-------- Reserved
 * ||||||||++--------- DMS source address space is DATA
 * |||||+++----------- SIND src address transfer index mode (++)
 * ||||+-------------- Reserved
 * |||+--------------- CTMOD Multiframe mode (not ABU)
 * ||+---------------- IMOD Interrupt at end of block
 * |+----------------- DINM Interrupt based on IMOD bit
 * +------------------ Autoinitialization disabled
 */
#define INIT_DMMCR4 ((u16)0x4141)

/* ------------------------------------------------------------------
 * Dma public methods
 * ------------------------------------------------------------------ */
/*
 * InitDma
 *
 *     This routine sets up McBSP2 for use with the
 * Codec.
 */
void InitDma(void)
{
  /*
   * Initialize DMPREC register
   */
   DMPREC=INIT_DMPREC;
  /*
   * Initialize Output channel 4 registers
   */
   DMSA=DMSRC4;
```

```
   DMSDI=INIT_DMSRC4;
   DMSDI=INIT_DMDST4;
   DMSDI=INIT_DMCTR4;
   DMSDI=INIT_DMSFC4;
   DMSDI=INIT_DMMCR4;
  /*
   * Initialize Input channel 5 registers
   */
   DMSA=DMSRC5;
   DMSDI=INIT_DMSRC5;
   DMSDI=INIT_DMDST5;
   DMSDI=INIT_DMCTR5;
   DMSDI=INIT_DMSFC5;
   DMSDI=INIT_DMMCR5;


}
/*
 * StartDma
 *
 *  Enable the input and output channels for the
 *  DMA. Obtain buffers for each channel.
 *  Unmask the DMA interrupts.
 */
void StartDma(void)
{
    BUF *temp;

/*
 * allocate buffers for the input and output channels
 * as well as the main processing loop
 */
   temp=AllocBuf();  // allocate a buffer
   PutBuf();         // put it to the process list
   temp=ReceiveBuf(); // get it for the calc position
   OutputBuf();       // put it into output list
   temp=GetBuf();     // retrieve it from the output list
   DMSA=DMSRC4;      // and make it the DMA's
   DMSDI=(u16)&temp->data;  // output buffer

   temp=AllocBuf();   // allocate a second buffer
   PutBuf();          // put it to the process list.
                      // by the time the output DMA
                      // finishes, this will be in
                      // the output buffer list
   temp=AllocBuf();   // get a third buffer
   DMSA=DMDST5;       // make it the DMA's
   DMSDI=(u16)&temp->data;   // input buffer

/*
 *    unmask the input and output interrupts
 */
   *IFR |= (u16)0x3000;
   *IMR |= (u16)0x3000;
/*
 * enable channels 4 and 5
 */
   DMPREC=DMPREC_ENABLE45;
}
/* ------------------------------------------------
 *  Private methods of DMA object
 * ------------------------------------------------ */
/*
 * DMAInInterrupt
 *     This routine is called by the interrupt routine
 * for DMA channel 5 (input channel)
 */
interrupt void DMAInInterrupt(void)
{
     BUF *inbuf;

    /*
     * Put the current input buffer to the
     * Processing buffer list.
     */
      PutBuf();
    /*
     * Allocate a new buffer for input
     */
     inbuf=AllocBuf();
     DMSA=DMDST5;
     DMSDI=(u16)&inbuf->data;
     DMSA=DMCTR5;
     DMSDI=(u16)(2*LEFTCHAN_SAMPLES-1);
     DMPREC |= DMPREC_ENABLE5;
}
/*
 * DMAOutInterrupt
 *     This routine is called by the interrupt routine
 * for DMA channel 4 (output channel)
 */
interrupt void DMAOutInterrupt(void)
{
```

```
     BUF *outbuf;

    /*
     * Free the current output buffer.
     */
     FreeBuf();
    /*
     * Get next buffer for output
     */
     outbuf=GetBuf();
     DMSA=DMSRC4;    // put out, set it up
     DMSDI=(u16)&outbuf->data;
     DMSA=DMCTR4;
     DMSDI=(u16)(2*LEFTCHAN_SAMPLES-1);
     DMPREC |= DMPREC_ENABLE4;
}

/*
 * dmaport.h
 *
 *     Header file for the DmaPort object
 *     Which is the McBsp2 serial port used
 *     by the DMA object.
 *
 */

#ifndef DMAPORT_H
#define DMAPORT_H
/*
 * McBSP2 serial port address and register sub-addresses
 * ( Used by dmaport.c and by dma.c )
 */
#define DRR2 ( * (volatile u16 *) 0x0030) // MCBSP2 Data Receive Reg 2
#define DRR1 ( * (volatile u16 *) 0x0031) // MCBSP2 Data Receive Reg 1
#define DXR2 ( * (volatile u16 *) 0x0032) // MCBSP2 Data Transmit Reg 2
#define DXR1 ( * (volatile u16 *) 0x0033) // MCBSP2 Data Transmit Reg 1

#define MCBSP2A ( * (volatile u16 *) 0x0034) // McBsp2 Subaddress Register
#define MCBSP2D ( * (volatile u16 *) 0x0035) // McBsp2 Data Registers
#define SPCR1  (0x0000)    // SPCR1 sub-address
#define SPCR2  (0x0001)    // SPCR2 sub-address
#define RCR1   (0x0002)    // RCR1 sub-address
#define RCR2   (0x0003)    // RCR2 sub-address
#define XCR1   (0x0004)    // XCR1 sub-address
#define XCR2   (0x0005)    // XCR2 sub-address
#define SRGR1  (0x0006)    // SRGR1 sub-address
#define SRGR2  (0x0007)    // SRGR2 sub-address
#define MCR1   (0x0008)    // MCR1 sub-address
#define MCR2   (0x0009)    // MCR2 sub-address
#define PCR    (0x000E)    // PCR sub-address
/*
 * public methods
 */
void InitDmaport (void);
void StartDmaport (void);
#endif

/*
 * dmaport.c
 *
 *    A voice substitution device module.
 *    Copyright 2003, David Clifton
 *    All Rights Reserved
 *
 *    data & methods for dmaport object
 *    a McBsp (2) which links Codec input/output
 *    to the DMA object.
 *
 */
#include "types.h"
#include "cpu.h"
#include "error.h"
#include "dmaport.h"
/*
 * local routine
 */
void putSampleOut(s16 out);

/*
 *
 * Set up McBSP2 for Format 0 of the Codec
 *
 * INPUT - 16 bit left justified
 *         sample on rising edge of bit clock
 *         rising edge delayed until middle data bit

 *
 * OUTPUT - 16 bit right justified
 *          sample on rising edge of bit clock
 *          rising edge delayed until middle of data bit

 *
 * SPCR1 bit field settings (RECEIVER = INPUT settings)
```

```
 *
 * 00100RR00001000
 * ||||||||||||||+-- RRST Receiver in Reset (initially)
 * |||||||||||||+--- RRDY Receiver ready status bit
 * ||||||||||||+---- RFULL Receiver overrun status bit
 * |||||||||||+----- RSYNCERR Detect sync error
 * ||||||||++------ RINTM  Interrupt on end of word
 * |||||||+-------- ABIS  abis mode disabled
 * ||||||+--------- DXENA DX mode disabled
 * |||||++---------- Reserved
 * |||++------------ CLKSTP Clock stop mode disabled
 * |++-------------- RJUST Right Justify, sign-extend MSBs in DRR[1,2]
 * +---------------- DLB Digital Loopback off
 */
#define INIT_SPCR1 (0x2008)
#define SPCR1_RXENABLE  (0x0001) // Put this into SPCR1 when ready to receive
/*
 * SPCR2 bit field settings
 *
 * RRRRRR0000000000
 * ||||||||||||||+-- XRST  Transmitter in Reset (initially)
 * |||||||||||||+--- XRDY  Transmitter ready status bit
 * ||||||||||||+---- XEMPTY Transmitter empty status bit
 * |||||||||||+----- XSYNCERR Transmitter sync eror bit
 * ||||||||||++------ XINTM  Transmit interrupt mode (end of word)
 * ||||||||+-------- GRST Sample rate generator is reset
 * |||||||+--------- FRST Frame synch generator is reset
 * ||||||+---------- SOFT Stop serial port immediately on breakpoint
 * |||||+----------- FREE Don't allow transmit to run on breakpoint
 * RRRRRR------------ Reserved
 */
#define INIT_SPCR2 (0x0000)
#define SPCR2_TXENABLE (0x0001)        // transmit enable
/*
 * definition of transmit ready bit in DSPCR2
 */
#define XRDY 0x0002

/*
 * PCR bit field settings
 *
 * RR000000RXXX0011
 * ||||||||||||||+-- CLKRP Sample on rising edge of  CLKR
 * |||||||||||||+--- CLKXP Transmit sample on falling edge CLKX
 * ||||||||||||+---- FSRP Receive Frame Synch (active high)
 * |||||||||||+----- FSXP  Transmit Frame Synch Polarity (active high)
 * ||||||||+++------ GP Receive Port Pin Values (irrelevant)
 * |||||||+--------- Reserved
 * ||||||+---------- CLKRM Receive Clock Mode (RX Clock is an input)
 * |||||+----------- CLKXM Transmit Clock Mode (TX Clock is an input)
 * ||||+------------ FSRM  Receive Frame Synch (Input)
 * |||+------------- FSXM  Transmit Frame Synch (Input)
 * ||+-------------- RIOEN Receive port pins enabled as serial port
 * ++--------------- XIOEN Transmit port pins enabled as serial port
 * ++--------------- Reserved
 */
#define INIT_PCR (0x0083)
/*
 * RCR1 bit field settings
 * (Just interested in left channel)
 *
 * R0000001010RRRRR
 * ||||||||||+++++---- Reserved
 * |||||||+++--------- RWDLEN1 Word length is 16 bits
 * |+++++++------------ RFRLEN1 Frame length is 2 word
 * +--------------------Reserved
 */
#define INIT_RCR1 (0x0140)
/*
 * RCR2 bit field settings
 *
 * 0000000101000000
 * ||||||||||||++---- RDATDLY No receive data deley
 * ||||||||||||+------ RFIG Frame sync pulses after first restart xfer
 * ||||||||||++------- RCOMPAND No companding
 * |||||||+++--------- RWDLEN2 Receive word length 2 is 16 bits (dont care)
 * |+++++++------------ RFRLEN2 Receive frame length 2 is 2 word (dont care)
 * +--------------------RPHASE Single phase frame
 */
#define INIT_RCR2 (0x0140)
/*
 * XCR1 bit field settings
 * (just interested in left channel)
 *
 * R0000001010RRRRR
 * ||||||||||+++++---- Reserved
 * |||||||+++--------- XWDLEN1 Word length is 16 bits
 * |+++++++------------ XFRLEN1 Frame length is 2 word
 * +--------------------Reserved
 */
#define INIT_XCR1 (0x0140)
/*
 * XCR2 bit field settings
```

```
 *
 * 0000000101000000
 * ||||||||||||||++---- XDATDLY No transmit data deley
 * |||||||||||||+------ XFIG Frame sync pulses after first restart xfer
 * ||||||||||++------- XCOMPAND No companding
 * ||||||||+++--------- XWDLEN2 Xmit word length 2 is 16 bits (dont care)
 * |+++++++------------ XFRLEN2 Xmit frame length 2 is 1 word (dont care)
 * +--------------------XPHASE Single phase frame
 */
#define INIT_XCR2 (0x0140)
/* ---------------------------------------------------------------------
 * Dmaport public methods
 * --------------------------------------------------------------------- */
/*
 * InitDmaport
 *
 *     This routine sets up McBSP2 for use with the
 * Codec.
 */
void InitDmaport(void)
{
  /*
   * Initialize McBsp2
   */
   MCBSP2A=SPCR1;
   MCBSP2D=INIT_SPCR1;   // Reset the McBSP2 transmitter
   MCBSP2A=SPCR2;
   MCBSP2D=INIT_SPCR2;   // Reset the McBSP2 receiver
   MCBSP2A=PCR;
   MCBSP2D=INIT_PCR;
   MCBSP2A=RCR1;
   MCBSP2D=INIT_RCR1;
   MCBSP2A=RCR2;
   MCBSP2D=INIT_RCR2;
   MCBSP2A=XCR1;
   MCBSP2D=INIT_XCR1;
   MCBSP2A=XCR2;
   MCBSP2D=INIT_XCR2;
}
/*
 * StartDmaport
 *
 *    Start up McBSP2 as the serial port which accepts data
 * from the Codec, and passes to the DMA, or accepts data
 * from the DMA and passes it to the codec.  Initial plan
 * is to Start this port only after the Codec and DMA
 * have been initialized to begin with.
 */
void StartDmaport(void)
{
/*
 * enable the receiver and transmitter
 */
   MCBSP2A=SPCR1;     // select the SPCR1 register
   MCBSP2D = (INIT_SPCR1 | SPCR1_RXENABLE);
   MCBSP2A=SPCR2;     // select the SPCR2 register
   MCBSP2D = (INIT_SPCR2 | SPCR2_TXENABLE);

   putSampleOut(0);

}

/* ---------------------------------------------------------------------
 * Dmaport private methods
 * --------------------------------------------------------------------- */
/*
 * putSampleOut
 *
 *     Wait for the transmit ready flag,
 * then put the sample supplied into the
 * output buffer of McBSP2.
 */
void putSampleOut(s16 out)
{
    u16 status=0;

    MCBSP2A=SPCR2;
    while((status & XRDY)==0) {
        status=MCBSP2D;
    }
    DXR1=out;
    DXR2=out;

}

/*
 * error.h
 *
 *     Header file for error object
 *
 */
#ifndef ERROR_H
#define ERROR_H
```

```
/*
 * Error codes
 */
typedef enum {
    NO_ERROR=0,
    SPURIOUS_INTERRUPT,
    SPURIOUS_TRAP,
    INBUF_BUSY,
    OUT_OF_BUFFERS,
    MISSING_INBUF,
    CALC_IN_PROGRESS,
    NO_BUF_TO_OUTPUT,
    NO_BUF_TO_FREE,
    STILL_USING_OUTPUT_BUF,
    BUFFER_SIZE_WRONG,
    NO_OUTPUT_BUFFERS,
    BAD_PROCESSING_STATE,
    MAXIMUM_ERROR_CODE
} ERRNUM;

/*
 * Public method definitions
 */
void InitError(void);
void HandleError(ERRNUM);
#endif

/*
 * error.c
 *
 * A Voice Substitution Device module
 * Copyright 2003, David Clifton
 * All Rights Reserved
 *
 * data & methods for error handling object
 */
#include "types.h"
#include "cpu.h"
#include "error.h"
/*
 * error code variable
 */
ERRNUM errcode=NO_ERROR;
/*-------------------------------------------------------------
 * Public method definitions
 * ------------------------------------------------------------- */
/*
 * InitError
 *
 *    Set the error code variable to zero
 */
void InitError(void)
{
    errcode=NO_ERROR;
}
/*
 * HandleError
 *
 *     Disable interrupts, set the error code to the value supplied,
 * and loop indefinitely.
 *
 */
void  HandleError(ERRNUM code)
{
    (void)DisableInterrupts();
    errcode=code;
    for(;;) {
       code=errcode;
    }
}

/*
 * filter.h
 *
 *     Header file for the Filter object
 *
 */
#ifndef FILTER_H
#define FILTER_H
/*
 * public methods
 */
void InitFilter (void);
void LowPass (s16 *);
#endif

/*
 * filter -- floating point filter
 *
 *    A voice substitution device module.
 *    Copyright 2003, David Clifton
 *    All Rights Reserved
 *
```

```
 *  This module does the filtering needed by the
 * voice period estimation modules.
 *
 */
#include "types.h"
#include "error.h"
#include "bufman.h"
#include "filter.h"

/*
 * The filter used here is a 4th order Butterworth,
 * with a cutoff of 0.015*sampling frequency,
 * having a gain per section of 0.00209109751.
 * Since the incoming frequency is
 * 48000Hz, the cutoff is 720Hz.
 *
 * Filter coefficients were computed using the
 * Digital Butterworth IIR Low Pass Design page
 * at www.nauticom.net/www/jdtaft/lobutter.htm
 *
 */
#define VECTOR_SIZE (SAMPLE_RATE/FRAME_RATE) // 240 samples in one buffer
/* ---------------------------------------------------
 *  local data
 * --------------------------------------------------- */
s16 *buf;        // pointer to input/output buffer
FLOATING y;      // intermediate value
/*
 * Variables for first Butterworth filter biquad.
 */
FLOATING b10=1.0;             // b coefficients
FLOATING b11=2.0;
FLOATING b12=1.0;

FLOATING a10=1.0;             // a coefficients
FLOATING a11=-1.8318539;      // .015 >>>>>>>>>
FLOATING a12=0.8400199;       // .015 >>>>>>>>>

FLOATING z11,z12;             // state variables
/*
 * Variables for second Butterworth filter biquad.
 */
FLOATING b20=1.0;              // b coefficients
FLOATING b21=2.0;
FLOATING b22=1.0;

FLOATING a20=1.0;             // a coefficients
FLOATING a21=-1.9219089;   // .015 >>>>>>>>>>>
FLOATING a22=0.9304764; // .015 >>>>>>>>>>>>
FLOATING z21,z22;             // state variables

/*
 * gain per section
 */
FLOATING g;
/* ---------------------------------------------------
 * Public methods
 * --------------------------------------------------- */
/*
 * InitFilter
 *
 *   Initialize the floating point filter
 */
void InitFilter(void)
{
    z11=0.0; // state variables
    z12=0.0;
    z21=0.0;
    z22=0.0;
    g=0.00209109751;  // gain per section
}
/*
 * LowPass
 *
 * Compute the results of applying the filter to
 * input vector supplied.  Place the results back into
 * the input vector.
 *
 */
void LowPass(s16 *x)
{
    s16 n;
    FLOATING d1,d2;

    for(n=0;n<VECTOR_SIZE;n++)
    {
      /*
       * compute first biquad
       */
        d1=g*x[n]-a11*z11-a12*z12;
        y=b10*d1+b11*z11+b12*z12;
        z12=z11;
        z11=d1;
```

```
      /*
       * compute second biquad
       */
        d2=g*y-a21*z21-a22*z22;
        y=b20*d2+b21*z21+b22*z22;
        z22=z21;
        z21=d2;

        x[n]=(s16)y;
    }

}

/*
 * pestr.h
 *
 *     Header file for the Pestr object
 */
#ifndef PESTR_H
#define PESTR_H
/*
 * limits on computed period
 * corresponding to a frequency range
 * of 50 to 960 Hz at a sampling
 * rate of 48kHz.
 */
#define MIN_PERIOD (48)
#define MAX_PERIOD (960)
/*
 * public methods
 */
void InitPestr(void);
void PestrNextSample(s16);
s16 GetPestr(void);
#endif

/*
 * pestr.c
 *
 *    A voice substitution device module.
 *    Copyright 2003, David Clifton
 *    All Rights Reserved
 *
 * This module maintains six different, time-based
 * pitch period estimators.  Three estimators are fed
 * with new measurements any time there is a peak
 * or a valley in the signal.
 * Each time it is queried, a pitch
 * estimator returns an estimate of pitch
 * period expressed as the number of
 * samples in the period.
 *
 *
 * Estimators work in the following way:
 *
 *  Each estimator has a time (sample count) and value
 *  for the last accepted peak(or valley).  Each time a new
 *  peak or valley entry is received, a decision is made whether
 *  or not to accept that entry.  The entry is accepted if its
 *  value exceeds an acceptance threshold.  It is rejected if
 *  it does not.
 *
 *  The acceptance threshold is computed from the value of
 *  the previously accepted entry and the time elapsed since
 *  that entry was accepted.  The acceptance threshold
 *  decays with time.  It equals the value accepted for
 *  a blanking period equal to 4/10ths of the current period
 *  estimate, and then decays exponentially with each
 *  successive sample period.
 *
 *  When an entry is accepted, the time, value and pitch
 *  period for that entry are updated.  The estimated pitch
 *  period is updated by averaging the time since the previous
 *  acceptance into a running average of the pitch period.
 *  If the entry is not accepted, the time, value, and pitch
 *  period estimates are unchanged.
 *
 *  For convenience, the current estimate has a maximum
 *  permissable value of 20 msec (960 samples), and a
 *  minimum permissable value of 1 msec (48 samples).
 *
 * For more information on this type of pitch estimation
 * see Theory and Applications of Digital Signal Processing,
 * Rabiner and Gold, pp681-687.
 */
#include <math.h>
#include "types.h"
#include "error.h"
#include "bufman.h"
#include "pestr.h"
/*
 * definition of period estimator type
 */
```

```
typedef struct pestr {
    u16 scount;   // sample counter of last accepted entry
    u16 value;    // value of last accepted entry
    u16 period;   // current pitch period estimate
} PESTR;
/*
 * limits of peak and valley values
 */
#define MAX_PEAK_VALUE (0x7FFF)
#define MIN_VALLEY_VALUE (0x8000)
/*
 * number of matches required for voicing decision
 */
#define MATCH_THRESHOLD (6)
/*
 * sample information
 */
u16 currScount;   // latest value of sample counter
s16 currValue;    // value of latest sample
s16 currDelta;    // difference between the latest
                  // sample, and the one before it
s16 prevValue;    // value of previous sample
s16 prevDelta;    // difference between the previous
                  // sample, and the one before it

/*
 * peak and valley information
 */
u16 peakScount;
u16 prevPeakScount;    // sample counter at latest peak
s16 peakValue;
s16 prevPeakValue;     // value of latest peak
u16 valleyScount;
u16 prevValleyScount;  // sample counter at latest valley
s16 valleyValue;
s16 prevValleyValue;   // value of latest valley

/*
 * array of six period estimators
 */
#define MAX_ESTIMATORS (6)
PESTR pest[MAX_ESTIMATORS];
/*
 * definition of each estimator
 */
#define PEAK_ENTRY (0)        // peak entry
#define P2PP_ENTRY (1)    // peak to previous peak entry
#define P2PV_ENTRY (2)    // peak to previous valley entry
#define VALLEY_ENTRY (3)     // valley entry
#define V2PP_ENTRY (4)    // valley to previous peak entry
#define V2PV_ENTRY (5)    // valley to previous valley entry
/*
 * historical period estimate array
 *    This array contains thirty six
 * period estimates, as follows.
 *
 * hpest[1][j], the first row, contains the six
 * most recent estimates from the estimators.
 * hpest[2][j], the second row, contains the
 * previous six extimates from the estimators.
 * hptest[3][j], the third row, contains the
 * six estimates prior to the previous six
 * estimates.
 * hpest[4][j], the fourth row, contains estimates
 * which are sums of the first and second rows.
 * hpest[5][j], the fifth row, contains estimates
 * which are sums of the second and third rows.
 * hpest[6][j], the sixth row, contains estimates
 * which are sums of the first three rows.
 */
#define MAX_ROWS (6)
u16 hpest[MAX_ROWS][MAX_ESTIMATORS]= {
   0,0,0,0,0,0,
   0,0,0,0,0,0,
   0,0,0,0,0,0,
   0,0,0,0,0,0,
   0,0,0,0,0,0,
   0,0,0,0,0,0
};
/*
 * negative exponential table lookup
 * This table contains negative exponentials of
 * numbers from 0 to 255.
 * Each number represents 100 times the value of
 * exp(-n/100), where n is the index into the
 * into the table.
 */
u16 etable[256];
/*
 * bogus period table
 *   Used to assign bogus value to period when
 * the value is zero
 */
```

```
u16 bogusPeriod=1;

/*
 * number of matches between each first row estimate
 * and the rest of the estimates in the hpest matrix
 */
u16 mcount[MAX_ESTIMATORS];
u16 mpeak;     // index of hpest[0] entry with most matches

/*
 * Private method declarations
 */
void updateEstimator(u16,u16);
u16 computeThreshold(PESTR *);
void updateEstimateHistory(void);
u16 countMatches(void);
u16 bestPest(void);
u16 nextBogusPeriod(void);

/* ------------------------------------------------
 * Public methods of period object
 * ------------------------------------------------ */
/*
 * InitPestr
 *
 *    Initialize peak and valley information. Set up
 * all the pitch period estimators with different initial
 * pitch periods and large thresholds.  This will prevent
 * the GetPestr() from concluding a voice is present until
 * the data has had a chance to accumulate.
 */
void InitPestr(void)
{
    s16 n;
    FLOATING x;

/*
 * initialize the exponential table
 */
    for(n=0;n<256;n++)
    {
        x=(FLOATING)n/100.0;
        etable[n]= (u16)(100.0*exp( -x  )+0.5);
    }
/*
 * initialize sample data
 */
    currScount=0;
    currValue=0;
    currDelta=0;
    prevValue=0;
    prevDelta=0;

/*
 * initialize peak and valley data
 */
    peakScount=0;    // sample counter at latest peak
    prevPeakScount=0xFFFF;  // sample counter at previous peak
    peakValue=MAX_PEAK_VALUE;     // value of latest peak
    prevPeakValue=MAX_PEAK_VALUE;   // value of previous peak
    valleyScount=0;  // sample counter at latest valley
    prevValleyScount=0xFFFE;
    valleyValue=MIN_VALLEY_VALUE;   // value of latest valley
    prevValleyValue=MIN_VALLEY_VALUE; // value of previous valley

/*
 * initialize peak period estimator
 */
    pest[PEAK_ENTRY].period=MIN_PERIOD;
    pest[PEAK_ENTRY].scount=0;
    pest[PEAK_ENTRY].value=MAX_PEAK_VALUE;

/*
 * initialize peak-to-prev-peak period estimator
 */
    pest[P2PP_ENTRY].period=80;
    pest[P2PP_ENTRY].scount=0;
    pest[P2PP_ENTRY].value=MAX_PEAK_VALUE;

/*
 * initialize up peak-to-prev-valley period estimator
 */
    pest[P2PV_ENTRY].period=150;
    pest[P2PV_ENTRY].scount=0;
    pest[P2PV_ENTRY].value=MAX_PEAK_VALUE;

/*
 * initialize valley period estimator
 */
    pest[VALLEY_ENTRY].period=210;
    pest[VALLEY_ENTRY].scount=0;
    pest[VALLEY_ENTRY].value=MAX_PEAK_VALUE;
/*
```

```
 * initialize valley-to-prev-peak period estimator
 */
    pest[V2PP_ENTRY].period=280;
    pest[V2PP_ENTRY].scount=0;
    pest[V2PP_ENTRY].value=MAX_PEAK_VALUE;
/*
 * initialize valley-to-prev-valley period estimator
 */
    pest[V2PV_ENTRY].period=MAX_PERIOD;
    pest[V2PV_ENTRY].scount=0;
    pest[V2PV_ENTRY].value=MAX_PEAK_VALUE;

}
/*
 * PestrNextSample
 *
 *    Called each time a new sample is passed to the period
 * estimators.  This routine rolls the sample data, and then
 * checks for a peak or valley.
 *
 *    If there is a peak, it rolls the peak data, and then
 * updates the three peak estimators.
 *
 *    If there is a valley, it rolls the valley data, and then
 * updates the three valley estimators.
 *
 * NOTE: a peak is a transition between a positive and
 * a negative delta that occurs with a positive sample value.
 * A valley is a transition between a negative and a positive
 * delta that occurs at a negative sample value.
 * Other transitions between negative and positive or positive
 * and negative deltas are ignored.
 */
void PestrNextSample(s16 sample)
{
    s16 temp;
   /*
    * roll sample information
    */
     currScount++;
     prevValue=currValue;
     currValue=sample;

     temp=currValue-prevValue;  // compute newest delta
   /*
    * don't roll the deltas unless the current value
    * differs from the previous value
    */
     if(temp!=0) {
         prevDelta=currDelta;   // roll
         currDelta=temp;        // deltas
       /*
        * check for a peak or valley
        */
        if(prevDelta>0 && currDelta<0 && sample > 0) {  // found a peak
          /*
           * roll the peak data
           */
            prevPeakScount=peakScount;
            prevPeakValue=peakValue;
            peakScount=currScount;
            peakValue=sample;
          /*
           * update the peak estimators
           */
            updateEstimator(PEAK_ENTRY,peakValue);
            temp=peakValue-prevPeakValue; // if the difference
            if(temp<0) temp= 0;           // is negative, use 0
            updateEstimator(P2PP_ENTRY,temp);
            updateEstimator(P2PV_ENTRY,(peakValue - prevValleyValue));
        } else if(prevDelta<0 && currDelta>0 && sample < 0) {  // found a valley
          /*
           * roll the valley data
           */
            prevValleyScount=valleyScount;
            prevValleyValue=valleyValue;
            valleyScount=currScount;
            valleyValue=sample;
          /*
           * update the valley estimators
           */
            updateEstimator(VALLEY_ENTRY,-1*valleyValue);
            updateEstimator(V2PP_ENTRY,(prevPeakValue-valleyValue));
            temp=-valleyValue+prevValleyValue;  // if the diff is neg
            if(temp<0) temp=0;                  // use zero
            updateEstimator(V2PV_ENTRY,temp);
        }
     }
}
/*
 * GetPestr
 *
 * Return the current period estimate (in number
```

```
 * of samples) from the Pestr object. This routine
 * compiles a matrix of recent estimates from the
 * period estimators, and uses a matching algorithm to
 * choose the most likely estimate of the period from
 * that matrix.  If there is not sufficient coincidence
 * to warrant a period estimate, it returns NIL,
 * indicating that the signal is currently
 * unvoiced. If there is sufficient coincidence in
 * the estimates, it returns an integer giving the
 * number of samples in the period estimate.
 */
s16 GetPestr(void)
{
    s16 numMatches;
    s16 result;

    updateEstimateHistory();
    numMatches=countMatches();
    if(numMatches>MATCH_THRESHOLD) {
        result = bestPest();
    } else {
        result = NIL;
    }
    return result;
}
/* ------------------------------------------------------------
 *  Private methods
 * ------------------------------------------------------------ */
/*
 * updateEstimator
 *
 * Compute the new threshold based upon the
 * estimator parameters and the current time.
 * Compare the new value (always positive) to the threshold.
 * If it is greater, update the estimator
 * with values from the new input.
 *     The first argument is the estimator index.
 * The second argument is the value supplied.
 */
void updateEstimator(u16 index,u16 sval)
{
    PESTR *ptr;
    u16 threshold,temp;
   /*
    * fill in the estimator fields
    */
    ptr=&pest[index];
    threshold=computeThreshold(ptr);

    if(sval>threshold) {
        temp= (ptr->period + (currScount - ptr->scount))/2;
        if(temp<MIN_PERIOD) temp=MIN_PERIOD;
        if(temp>MAX_PERIOD) temp=MAX_PERIOD;
        ptr->period =temp;
        ptr->value = sval;
        ptr->scount= currScount;
    } else if(sval==0) {
        ptr->period=nextBogusPeriod();
        ptr->value = sval;
        ptr->scount = currScount;
    }
}
/*
 * computeThreshold
 *
 * Given a pointer to a period estimator,
 * compute the threshold value beyond which
 * estimator should be updated.
 *
 * The threshold is computed from the previous
 * threshold value, and the time that has elapsed
 * since that value was set.
 *
 * If the time difference between the initial
 * time and the current time is less than a
 * blanking period, the threshold is just
 * same as the initial value.
 *
 * If the time difference is greater than the
 * blanking period, the threshold is a fraction
 * of the initial value.  The size of the fraction
 * is computed as exp(-0.695*elapsedTime/period)
 *
 */

u16 computeThreshold(PESTR *ptr)
{
    u16 j;

    u16 blankingInterval;
    u16 elapsedTime;
    u16 threshold;
    FLOATING thr;
```

```
    blankingInterval=4*ptr->period/10;
    elapsedTime= currScount-ptr->scount;
    if(elapsedTime<blankingInterval) {
        threshold=0xFFFF;
    } else {
/*
 * For speed, the two executable lines below this comment
 * are replaced with the third executable line below
 * this comment.
 */
#if 0
        beta=(FLOATING)(ptr->period)/0.695;
        j=(u16)(100*elapsedTime/beta);
#else
        j=(u16)(70*elapsedTime/ptr->period); // replace 2 lines above for speed
#endif
        if(j>255) {
            threshold=ptr->value/10;
        } else {
            thr= (FLOATING)(ptr->value)*etable[j]/100.0;
            threshold=(u16)thr;
        }
    }
    return threshold;
}
/*
 * updateEstimateHistory
 *
 *    This routine copies the second pest[] row
 * to the third, and the first to the second.
 * Then it fills in the first row with current
 * values from the period estimators.  Finally,
 * it computes the last three rows from the
 * first three.
 */
void updateEstimateHistory(void)
{
    s16 j;

    for(j=0;j<6;j++) {
         hpest[2][j]=hpest[1][j];
         hpest[1][j]=hpest[0][j];
         hpest[0][j]=pest[j].period;
         hpest[3][j]=hpest[0][j]+hpest[1][j];
         hpest[4][j]=hpest[1][j]+hpest[2][j];
         hpest[5][j]=hpest[3][j]+hpest[2][j];
    }
}
/*
 * countMatches
 *
 *    This routine looks at the ratios of the
 * entries in the first hpest row, with the
 * entries in the rest of the matrix.  For each
 * entry in the first row, it counts the number
 * of entries in the second through the sixth
 * rows with with a ratio between .98 and 1.02.
 * Returns the total matches found.
 */
u16 countMatches(void)
{
    u32 ratio;
    u16 dapest;
    s16 j,k,l;
    s16 total=0;

    for(j=0;j<MAX_ESTIMATORS;j++) { // for each first row entry
        mcount[j]=0;                // clear match count
        for(k=1;k<MAX_ROWS;k++) {   // for each row
           for(l=0;l<MAX_ESTIMATORS;l++) {
              ratio=(u32)hpest[0][j];
                    ratio*=100;
                    dapest=hpest[k][l];
                    ratio/=dapest;
                    if(ratio>=98 && ratio<=102) {
                         mcount[j]+=1;
                         total+=1;
                    }
           }
        }
    }
    return total;
}
/*
 * bestPest
 *    This routine selects the entry in the first
 * row of hpest which has the most matches.  It returns
 * the period estimate in that entry.
 */
u16 bestPest(void)
{
     u16 j;
```

```
     u16 mindex=0;
     u16 temp=0;

     for(j=0;j<MAX_ESTIMATORS;j++) {
         if(mcount[j]>temp) {
             temp=mcount[j];
             mindex=j;
         }
     }
     temp=hpest[0][mindex];
     return temp;
}
/*
 * nextBogusPeriod
 *
 */
u16 nextBogusPeriod(void)
{
    bogusPeriod+=17;
    bogusPeriod%=19;
    return bogusPeriod;
}

/*
 * procman.h
 *
 *     Header file for the Procman object
 *
 */

#ifndef PROCMAN_H
#define PROCMAN_H

/*
 * public methods
 */
void InitProcess(void);
void Process(void);
#endif

/*
 * procman.c
 *
 *    A voice substitution device module.
 *    Copyright 2003, David Clifton
 *    All Rights Reserved
 *
 * This module coordinates the processing of audio
 * buffers with the setting of the processing parameters
 * of waveform and volume.
 */
#include "types.h"
#include "error.h"
#include "auxport.h"
#include "codec.h"
#include "bufman.h"
#include "calc.h"
#include "tone.h"
#include "procman.h"
/*
 * procman states
 */
typedef enum { PSTART=NIL,PIDLE=0,PROCESSING,PCLEANUP } PSTATE;
/*
 * procman data
 */
PSTATE pstate=PSTART;    // current processing state
BUF *currentBuf=NULL;    // current buffer being processed

/* ------------------------------------------------
 * private method declarations
 * ------------------------------------------------ */
void idleState(void);
void processingState(void);
void cleanupState(void);
u16 potToVol(u16);
void handleInputs(void);

/* ------------------------------------------------
 * public methods
 * ------------------------------------------------ */
/*
 * InitProcess
 *    Initializes the procman object by initializing
 * its processing state.
 */
void InitProcess(void)
{
    pstate=PIDLE;
}
/*
 * Process
 *     Execute the process state machine.
```

```
 */
void Process(void)
{
    switch(pstate) {
    case PSTART:
        break;
    case PIDLE:
        idleState();
        break;
    case PROCESSING:
        processingState();
        break;
    case PCLEANUP:
        cleanupState();
        break;
    default:
        HandleError(BAD_PROCESSING_STATE);
    }
}
/* ------------------------------------------------------
 *  private methods
 * ------------------------------------------------------ */
/*
 * idleState
 *    If there is a new buffer to process, check for a new
 * waveform and volume.  If there is a new waveform, convey
 * it to the tone object.  If there is a new volume, pass it
 * to the codec object. Set the currentBuf pointer and
 * change state to process.
 */
void idleState(void)
{

    currentBuf=ReceiveBuf();
    if(currentBuf!=NULL) {  // got a new buffer to process
        handleInputs();
        pstate=PROCESSING;
    }
}
/*
 * processingState
 *     Call Calc() until it returns 0.  Then
 * change state to cleanup.
 */
void processingState(void)
{
    if(Calc(currentBuf)==0) {
        pstate=PCLEANUP;
    }
}
/*
 * cleanupState
 *    Call OutputBuf() for the buffer that just
 * finished processing.  Clear the currentBuf
 * pointer and set the state to IDLE.
 */
void cleanupState(void)
{
    OutputBuf();
    currentBuf=NULL;
    pstate=PIDLE;
}
/*
 * potToVol
 *   Convert the potentiometer reading (0-MAX_POTVAL)
 * to a volume reading (0-255)
 */
u16 potToVol(u16 pot)
{
     u32 temp;
     u16 vol;

     temp=(u32)pot;
     vol=(u16)(temp*255/MAX_POTVAL);

     return vol;
}

/*
 * handleInputs
 *    Check for button and potentiometer inputs
 * and reset waveform and volume as needed.
 */
void handleInputs(void)
{
    if(PotChanged()) {
        SetVolume(potToVol(LatestPot()));
    }
    if(SinePressed()) {
        SetWaveform(SINE);
    } else if(TrianglePressed()) {
        SetWaveform(TRIANGLE);
    } else if(SawPressed()) {
```

```
        SetWaveform(SAW);
    } else if(SquarePressed()) {
        SetWaveform(SQUARE);
    }
}

/*
 * tone.h
 *
 *     Header file for the Tone object
 *
 */
#ifndef TONE_H
#define TONE_H
/*
 * wave type definitions
 */
typedef enum {SINE=0,TRIANGLE,SAW,SQUARE} WAVE;
/*
 * public methods
 */
void InitTone(void);
void SetWaveform(WAVE);
void ToneInject(BUF *,u16,u16);
#endif

/*
 * tone.c
 *
 *    A voice substitution device module.
 *    Copyright 2003, David Clifton
 *    All Rights Reserved
 *
 * This module generates the tones which are injected
 * into the left channel of an output buffer when
 * voicing is found to be present in the input
 * signal.
 */

#include <math.h>
#include "types.h"
#include "error.h"
#include "codec.h"
#include "bufman.h"
#include "pestr.h"
#include "tone.h"
/*
 * tone phase
 *    phase varies from 0 to 65535 over the
 *  length of a single cycle.
 */
#define TABLE_CYCLE ((u32)256)   // 256 samples in a waveform table cycle
u16 phase;                       // 0 to 65535
/*
 * waveform tables -- tables with 256 wave samples having
 * amplitude of 128
 */
s16 squareTable[256];
s16 sawTable[256];
s16 sineTable[256];
s16 triangleTable[256];
/*
 * current waveform
 */
WAVE w;
/* ------------------------------------------------
 * private methods
 * ------------------------------------------------ */
void waveInject(s16 *,BUF *,u16,u16);
/*
 * InitTone
 *    Set the phase to zero.zero
 */
void InitTone(void)
{
    s16 j;
  /*
   * initialize sine table
   */
    for(j=0;j<256;j++) {
        sineTable[j]=(s16)(128.0*sin(2.0*PI*(FLOATING)j/256.0));
    }
  /*
   * initialize triangle table
   */
    for(j=0;j<256;j++) {
       if(j<64) {
           triangleTable[j]=(s16)(2*j);
       } else if(j<192){
           triangleTable[j]=(s16)(256 - 2*j);
       } else if(j<256) {
           triangleTable[j]=(s16)(2*j-512);
       }
```

```
    }
  /*
   * initialize saw table
   */
    for(j=0;j<256;j++) {
        sawTable[j]=(s16)(128 - j);
    }
  /*
   * initialize square table
   */
    for(j=0;j<256;j++) {
        if(j<128) {
            squareTable[j]=(s16)128;
        } else {
            squareTable[j]=(s16)-128;
        }
    }
  /*
   * initialize phase to 0
   */
    phase=0;
  /*
   * default waveform is saw
   */
    w=SAW;
}
/*
 * SetWaveform
 *
 * Set the voice waveform to the one supplied in the
 * single argument.
 */
void SetWaveform(WAVE wf)
{
    w=wf;
}
/*
 * ToneInject
 *
 * Inject into the buffer provided in the second argument,
 * a wave form indicated in the first argument, with amplitude
 * provided in the third argument, frequency provided in the
 * fourth argument, and starting phase provided in the local
 * variable phase.  Update the phase variable for each sample.
 *
 * freq is in Hz.
 *
 * phase varies from 0 to 65535 from the beginning to
 * the end of a cycle.
 *
 */
void ToneInject(BUF *bptr,u16 vol ,u16 freq)
{
   switch(w) {
   case SINE:
       waveInject(sineTable,bptr,vol,freq);
       break;
   case TRIANGLE:
       waveInject(triangleTable,bptr,vol,freq);
       break;
   case SAW:
       waveInject(sawTable,bptr,vol,freq);
       break;
   case SQUARE:
       waveInject(squareTable,bptr,vol,freq);
       break;
   }
}
/* ------------------------------------------------------------
 *  Private methods
 * ------------------------------------------------------------ */
/*
 * waveInject
 *
 *     Inject into the output buffer supplied the wave
 * whose wave table is supplied, with the frequency
 * and amplitude supplied.
 */
void waveInject(s16 *wtbl,BUF *bptr,u16 vol ,u16 freq)
{
    u16 j,k,M;
    u16 pdelta;
    u32 work;
    s16 wav;
    s32 temp;
/*
 * compute the number of samples in one cycle
 * of the frequency supplied
 */
    M= ((u16)SAMPLE_RATE/freq);
    pdelta=(u16)((u32)0x10000/M);
/*
 * fill buffer with wave starting with
```

```
 * appropriate phase
 */
  for(j=0;j<LEFTCHAN_SAMPLES;j++)
  {
      work=(u32)phase*TABLE_CYCLE;
      k=(u16)(work/0x10000);
      temp=(s32)wtbl[k]*(s32)vol/128;
      wav=(s16)temp;
      bptr->data[2*j]=wav;
      bptr->data[2*j+1]=wav;
      phase += pdelta;
  }
}

/*
 * voice.h
 *
 *     Header file for the Voice object
 */
#ifndef VOICE_H
#define VOICE_H
/*
 * public methods
 */
void InitVoice(void);
s16 CopyBuffer(BUF *);
void FilterBuffer(void);
u16 EstimatePitch(void);
#endif

/*
 * voice.c
 *
 *    A voice substitution device module.
 *    Copyright 2003, David Clifton
 *    All Rights Reserved
 *
 * This module makes a local copy of the
 * buffer supplied, and low pass filters
 * the data in that copy.  Then it performs
 * one or more analyses to determine whether
 * the buffer contains voiced or unvoiced
 * sound, and if voiced, determines the fundamental
 * pitch frequency.
 *
 *
 */
#include "types.h"
#include "error.h"
#include "bufman.h"
#include "filter.h"
#include "voice.h"
/*
 * local data
 */
s16 avgAbsMag;  // average absolute magniture of the signal
                // in the buffer provided to the CopyBuffer
                // routine.
s16 leftChan[LEFTCHAN_SAMPLES];
/* -------------------------------------------------------
 *  private method declarations
 * ------------------------------------------------------- */
/* -------------------------------------------------------
 *  public methods
 * ------------------------------------------------------- */
/*
 * InitVoice
 *
 * Initialize the voice period estimator routine(s)
 */
void InitVoice(void)
{
   ;   // nothing needed here
}
/*
 * CopyBuffer
 *
 *    This routine copies the data from the buffer supplied
 * into a buffer which contains a single 16 bit value per
 * sample.  As it copies the buffer, it computes the average
 * absolute magnitude of the signal. Then it performs a low pass filter operation
 * on the contents of the buffer, leaving the result in the
 * same buffer.  The average absolute magnitude is returned
 * to the caller.
 *
 *    This should be much easier than it is, but
 * with breakpoints in the code, sometimes the left channel
 * ends up in the even numbered buffer words, and sometimes it ends
 * up in the odd numbered buffer words.
 *
 *    Rather than track down and handle synchronization
 * problems between the Dma and McBsp2 in the presence of
 * breakpoints , it was just easier to handle both conditions
```

```
 * in this routine.
 */
s16 CopyBuffer(BUF *bptr)
{
    s16 j,mag1,mag2;
    s32 sumOfAbsVals=0;

    for(j=0;j<LEFTCHAN_SAMPLES;j++) {
        mag1=bptr->data[2*j];
        mag2=bptr->data[2*j+1];
        if(mag1==0) {      // left channel in odd numbered words
            if(mag2<0) {
               sumOfAbsVals-=mag2;
            } else {
               sumOfAbsVals+=mag2;
            }
            bptr->data[2*j]=mag2;
            bptr->data[2*j+1]=mag2;
            leftChan[j]=mag2;
        } else {           // left channel in even numbered words
            if(mag1<0) {
               sumOfAbsVals-=mag1;
            } else {
               sumOfAbsVals+=mag1;
            }
            bptr->data[2*j]=mag1;
            bptr->data[2*j+1]=mag1;
            leftChan[j]=mag1;
        }

    }
    avgAbsMag=sumOfAbsVals/(LEFTCHAN_SAMPLES);
    return avgAbsMag;
}
/*
 * FilterBuffer
 *
 *   This routine calls the routine LowPass()
 * in the filter module to low pass filter in place
 * the contents of the left channel buffer.
 */
void FilterBuffer(void)
{
    LowPass(leftChan);
}
/*
 * EstimatePitch
 *
 *     This routine passes each filtered sample
 * in the left channel buffer to a pitch estimating
 * routine in the module pestr.  Then it calls the
 * routine GetPestr() to obtain the latest estimate
 * of the voiced/unvoiced decision and the pitch frequency.
 *
 *     It returns an integer to the caller which is
 * NIL if the buffer is predominately unvoiced, and
 * equal to the pitch frequency if it is predominately
 * voiced.
 */
u16 EstimatePitch(void)
{
    s16 j;

    for(j=0;j<LEFTCHAN_SAMPLES;j++)
    {
        PestrNextSample(leftChan[j]);
    }
    j=GetPestr();
    if(j!=NIL) {
        j=SAMPLE_RATE/j;
    }
    return j;
}

/***************************************************************************/
/*                                                                         */
/* LNK.CMD - V2.00  COMMAND FILE FOR LINKING C PROGRAMS                    */
/*                                                                         */
/*      Usage:  lnk500 <obj files...>    -o <out file> -m <map file> lnk.cmd */
/*              cl500  <src files...> -z -o <out file> -m <map file> lnk.cmd */
/*                                                                         */
/*      Description: This file is a sample command file that can be used     */
/*                   for linking programs built with the C54x C Compiler.    */
/*                   This file has been designed to work for                 */
/*                   548 C54x device.                                        */
/*                   Use it as a guideline; you may want to make alterations */
/*                   appropriate for the memory layout of the target         */
/*                   system and/or your application.                         */
/*                                                                           */
/*      Notes: (1)   You must specify the directory in which rts.lib is      */
/*                   located.  Either add a "-i<directory>" line to this     */
/*                   file, or use the system environment variable C_DIR to   */
/*                   specify a search path for the libraries.                */
```

```
/*                                                                          */
/*             (2)   If the run-time library you are using is not           */
/*                   named rts.lib, be sure to use the correct name here.   */
/****************************************************************************/

/* choose initialization strategy (-c or -cr): */
/*-cr     downloader initializes global variables (not cold boot)   */

-c  /* copy constants in .cinit to global variables at boot time */
-x  /* */

-ovsd.out

-mvsd.map

-id:\ti\c5400\cgtools\lib

 -l rts_ext.lib

/* specify sizes of stack & heap: */
-stack 0x800 /* size of .stack section */
/* -heap  0x800 /* size of .sysmem section */

MEMORY {
   PAGE 0: /* program memory */

      /* IF       THIS IS                              */
      /* ===      =======                              */
      /* OVLY=1   some of internal RAM and/or external  */
      /*          RAM depending on which C54x this is.  */
      /* OVLY=0   external RAM                         */
          PROG_RAM (RWX) : origin = 0x5400, length = 0x2B80
          PROG_EXT (RWX) : origin = 0x8000, length = 0x4000

      /* boot interrupt vector table location */
        VECTORS (RWX): origin = 0x7F80, length = 0x80


   PAGE 1: /* data memory, addresses 0-7Fh are reserved */

      /* some (or all) of internal DARAM */
          DATA_RAM (RW): origin = 0x80, length = 0x5380
          DATA_EXT (RW): origin = 0x8000, length = 0x7FFF


   PAGE 2: /* I/O memory */

      /* no devices declared */

} /* MEMORY */


SECTIONS {
   .text    >> PROG_RAM | PROG_EXT PAGE 0   /* code                     */
   .switch  > PROG_RAM PAGE 0               /* switch table info        */

/* .cinit must be allocated to PAGE 0 when using -c (vs -cr)            */
   .cinit   > PROG_RAM PAGE 0  /* commented because -cr used above     */

   .vectors > VECTORS PAGE 0               /* interrupt vectors         */

   .cio     >  DATA_RAM PAGE 1             /* C I/O                     */
   .data    >> DATA_RAM | DATA_EXT PAGE 1  /* initialized data          */
   .bss     >> DATA_RAM | DATA_EXT PAGE 1  /* global & static variables */
   .const   >  DATA_RAM PAGE 1             /* constant data             */
   .sysmem  >> DATA_RAM | DATA_EXT PAGE 1  /* heap                      */
   .stack   >> DATA_RAM | DATA_EXT PAGE 1  /* stack                     */
} /* SECTIONS */
```

BIBLIOGRAPHY

Alhir, Sinan Si, UML In A Nutshell, O'Reilly,1998.

Boehm, Barry W et al, Software Cost Estimation with Cocomo II, Prentice Hall PTR, 2000

Booch, Grady, Object Oriented Design with Applications, Benjamin/Cummings Publishing Company, Inc, 1991.

Booch, Rumbaugh, and Jacobson, The Unified Modeling Language User Guide, Addison Wesley, 2000.

Burr Brown Corporation, 16-/20-Bit Single-Ended Analog Input/Output SoundPlus Stereo Audio CODECs, PCM3002, PCM3003, 1997.

Coad, Peter and Yourdon, Edward, Object Oriented Analysis, Prentice Hall PTR, 1990.

Demarco, Tom, Structured Analysis and System Specification, Yourdon Press, 1985.

Garmus, David and Herron, David, Function Point Analysis, Addison Wesley, 2001.

B. Gold and L.R. Rabiner, "Parallel Processing Techniques for Estimating Pitch Periods of Speech in the Time Domain," J. Acoust. Soc. Am., **46**, No. 2, 442-449, Aug 1969.

Hatley, Derek et al, Strategies for Real Time System Specification, Dorset House, 1988.

Jacobson, Ivar, Object Oriented Software Engineering, Addison Wesley, 1992.

Leffingwell, Dean and Widrig, Don, Managing Software Requirements: A Unified Approach, Addison Wesley, 1999.

Parallax Inc., BASIC Stamp Activity Board 27905, Features and Specifications, www.parallaxinc.com.

Rabiner, Lawrence R. and Gold, Bernard, Theory and Application of Digital Signal Processing, Prentice-Hall, Inc 1975.

Spectrum Digital Incorporated, TMS320VC5416DSK Technical Reference, 506005-0001 Rev A March 2002.

Synthesizers.com, http://www.synthesizers.com

Tesla Society of New York, http://www.teslasociety.com